MASTER EN SCIENCES INFORMATIQUES

Prototypage en OBLOG de spécifications de cahiers des charges en ALBERT II

Léonard, Laurent

Award date:
1996

Awarding institution:
Universite de Namur

Facultés Universitaires
Notre-Dame de la Paix
Institut d'Informatique
Namur

Prototypage en OBLOG de spécifications de cahiers des charges en ALBERT II.

Laurent LEONARD

Promoteur : Professeur Eric DUBOIS

Mémoire présenté en vue de l'obtention du grade de Maître en Informatique
Année Académique 1995-1996

LBS 6844094

307333

EXEMPLE

ENONCE.

La fabrique de boulons est dirigée par un ouvrier qui contrôle une machine. Le rôle de la fabrique est de produire des boulons à partir de rivets en y traçant un filet.

La machine possède trois bacs : le bac d'entrée contenant les rivets (capacité maximum de 200), le bac de sortie (capacité maximum de 200) où sont stockés les boulons produits et le bac de rebut où sont éjectés les boulons non terminés en cas de problème. La machine ne peut pas fonctionner et ignore les ordres de l'ouvrier lorsque le bac d'entrée est vide ou lorsque le bac de sortie est plein. L'ouvrier est chargé de remplir le bac d'entrée de rivets et de sortir les boulons produits du bac de sortie.

La machine est dotée de deux boutons que peut actionner l'ouvrier : un bouton rouge et un bouton vert. L'ouvrier enfonce le bouton vert pour demander à la machine la production d'un nouveau boulon. Quand l'ouvrier appuie sur le bouton rouge, le boulon en cours de fabrication est éjecté dans le bac de rebut.

La production d'un boulon consiste à répéter 120 fois le traçage d'un filet à l'intérieur du rivet.

La machine allume un témoin lorsque le bac de sortie est plein mais l'ouvrier, n'étant pas toujours attentif, ne s'en rend pas toujours compte.

Le bac d'entrée contient au départ 100 rivets, les bacs de sortie et de rebut sont vides.

AGENT MACHINE

STATE COMPONENTS

NbrRivets **instance of** INTEGER → OUVRIER
NbrBoulons **instance of** INTEGER → OUVRIER
NbrRebuts **instance of** INTEGER → OUVRIER
NbrMaxRivets **instance of** INTEGER*
NbrMaxBoulons **instance of** INTEGER*
BacEntreeVide **derived from** NbrRivets
instance of BOOLEAN
BacSortiePlein **derived from** NbrBoulons, NbrMaxBoulons
instance of BOOLEAN
Occupe **instance of** BOOLEAN
Temoin **instance of** BOOLEAN → OUVRIER

ACTION

PrendreRivet*
TracerFilet*
DeposerRebut*
DeposerBoulon*
AllumerTemoin
EteindreTemoin
Produire
TracerTousFilets*
Rejeter
TracerPartieFilets*
TracerFiletOuRien*
Avorter*

BASIC CONSTRAINTS

DERIVED COMPONENTS

BacEntreeVide ≡ NbrRivets = 0
BacSortiePlein ≡ NbrBoulons = NbrMaxBoulons

INITIAL VALUATION

NbrRivets = 100
NbrBoulons = 0
NbrMaxBoulons = 200
NbrMaxRivets = 200
Occupe = FALSE
Temoin = FALSE

Local Constraints

State Behaviour

$NbrRivets \geq 0$
$NbrBoulons \geq 0$
$NbrRebuts \geq 0$

Effects of Actions

PrendreRivet : NbrRivets := NbrRivets - 1
DeposerRebut : NbrRebuts := NbrRebuts + 1 ; Occupe := FALSE
DeposerBoulon : NbrBoulons := NbrBoulons + 1 ; Occupe := FALSE
AllumerTemoin : Temoin := TRUE
EteindreTemoin : Temoin := FALSE
Ouvrier.RetirerBoulons : NbrBoulons := 0
Ouvrier.MettreRivets(n) : NbrRivets := NbrRivets + n
Ouvrier.PresserVert : Occupe := TRUE

Capacity

$F(\ Produire \mid BacEntreeVide \vee BacSortiePlein\ )$
$F(\ Rejeter \mid BacEntreeVide \vee BacSortiePlein\ )$
$XO(\ AllumerTemoin \mid \neg Temoin \wedge BacSortiePlein\ )$
$XO(\ EteindreTemoin \mid Temoin \wedge \neg BacSortiePlein)$

Action Composition

$Produire \leftrightarrow Ouvrier.PresserVert\ ;\ PrendreRivet\ ;\ TracerTousFilets\ ;$
$DeposerBoulon$
$TracerTousFilets \leftrightarrow \{TracerFilet\}^{120}$
$Rejeter \leftrightarrow Ouvrier.PresserVert\ ;\ PrendreRivet\ ;$
$TracerPartieFilets\ ;$
$Avorter$
$TracerPartieFilets \leftrightarrow \{TracerFiletOuRien\}^{120}$
$TracerFiletOuRien \leftrightarrow TracerFilet \oplus DAC$
$Avorter \leftrightarrow Ouvrier.PresserRouge \otimes DeposerRebut$

Action Duration

$|PrendreRivets| = 0\ sec$
$|TracerFilet| = 0\ sec$
$|DeposerRebut| = 0\ sec$
$|DeposerBoulon| = 0\ sec$
$|AllumerTemoin| = 0\ sec$
$|EteindreTemoin| = 0\ sec$

Cooperation Constraints

Action Perception

XK(O.PresserVert | ⌉ Occupe ∧ ⌉ BacEntreeVide ∧ ⌉ BacSortiePlein)
XK(O.PresserRouge | Occupe)
K(O.RetirerBoulons | TRUE)
K(O.MettreRivets(n) | TRUE)

State Information

K(NbrRivets.Ouvrier | TRUE)
K(NbrBoulons.Ouvrier | TRUE)
K(NbrRebuts.Ouvrier | TRUE)
K(Temoin.Ouvrier | TRUE)

Agent Ouvrier

ACTIONS

MettreRivets(INTEGER)	→	MACHINE
RetirerBoulons	→	MACHINE
PresserVert	→	MACHINE*
PresserRouge	→	MACHINE*

Cooperation Constraints

State Perception

K(Machine.NbrRivets | TRUE)
K(Machine.NbrBoulons | TRUE)
K(Machine.NbrRebuts | TRUE)
K(Machine.Temoin | TRUE)
K(Machine.BacEntreeVide | TRUE)
K(Machine.BacSortiePlein | TRUE)

Action Information

K(PresserVert.Machine | TRUE)
K(PresserRouge.Machine | TRUE)
K(RetirerBoulon.Machine | TRUE)
K(MettreRivets.Machine | TRUE)

TRADUCTION

SUPPRESSION DE L'AGENT OUVRIER.

L'agent MACHINE hérite des actions :

- `MettreRivets (INT)`
- `RetirerBoulons`
- `PresserVert`
- `PresserRouge`

IDENTIFICATION DES AGENTS QUI AFFICHENT OU SAISISSENT DES DONNEES.

L'agent MACHINE affiche des données et en saisit. Il sera donc du type DBX.

TRANSFORMATION DES AGENTS EN OBJETS & AJOUT DES ACTIONS DE NAISSANCE ET DE MORT.

```
Single DBX Class Machine

Attributes
NbrRivets                    : Int
NbrBoulons                   : Int
NbrRebuts                    : Int
NbrMaxRivets                 : Int
NbrMaxBoulons                : Int
BacEntreeVide                : Bool
BacSortiePlein               : Bool
NbrFilets                    : Int[1]
Occupe                       : Bool
Temoin                       : Bool

Operations
PrendreRivets
TracerFilet
DeposerRebut
DeposerBoulon
AllumerTemoin
EteindreTemoin
```

[1] L'attribut `NbrFilets` est ajouté pour compté le nombre de filets déjà tracés.

```
MettreRivets
RetirerBoulons
PresserVert[2]

Init[3]

*Naitre
+Mourir


Operation Naitre
     Updates
     NbrRivets        := 100
     NbrBoulons       := 0
     NbrMaxBoulons    := 200
     NbrMaxRivets     := 200
     Temoin           := FALSE
     Occupe           := FALSE
     BacEntreeVide    := NbrRivets = 0
     BacSortiePlein   := NbrBoulons = NbrMaxBoulons

Operation PrendreRivet
     Updates
     NbrRivets        := NbrRivets - 1
     BacEntreeVide    := NbrRivets = 0

Operation DeposerRebut
     Updates
     NbrRebuts        := NbrRebuts + 1
     Occupe           := FALSE

Operation DeposerBoulon
     Updates
     NbrBoulons       := NbrBoulons + 1
     Occupe           := FALSE
     BacSortiePlein   := NbrBoulons = NbrMaxBoulons

Operation AllumerTemoin
     Updates
     Temoin           := TRUE

Operation EteindreTemoin
     Updates
     Temoin           := FALSE

Operation RetirerBoulons
     Updates
     NbrBoulons       := 0
     BacSortiePlein   := NbrBoulons = NbrMaxBoulons

Operation MettreRivets
```

---

[2] L'action `PresserRouge` est reprise dans l'action `DeposerRebut` parce qu'elles ont un déroulement en parallèle.

[3] L'action `Init` est ajoutée pour gérer l'occurrence répétitive de l'action `TracerFilet`.

```
     Parameters
     n                 : Int
     Updates
     NbrRivets         := NbrRivets + MettreRivets.n
     BacEntreeVide     := NbrRivets = 0

Operation PresserVert
     Updates
     Occupe            := TRUE

Operation Init
     Updates
     NbrFilets         := 1
Operation TracerFilet
     Updates
     NbrFilets         := NbrFilets + 1
```

CONSTRUCTION DE L'OBJET OBSERVATEUR.

```
Single DBX Class Observateur

Attributes
NbrRivets                       : Int
NbrBoulons                      : Int
NbrRebuts                       : Int
NbrMaxRivets                    : Int
NbrMaxBoulons                   : Int
BacEntreeVide                   : Bool
BacSortiePlein                  : Bool
Occupe                          : Bool
Temoin                          : Bool

Operations
MettreAJour
*Naitre
+Mourir
```

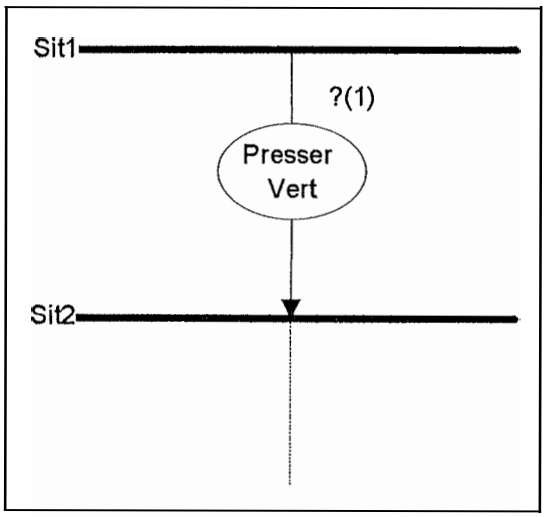
...

CONSTRUCTION DES GRAPHES DE COMPORTEMENT.

TRADUCTION LOCALE.

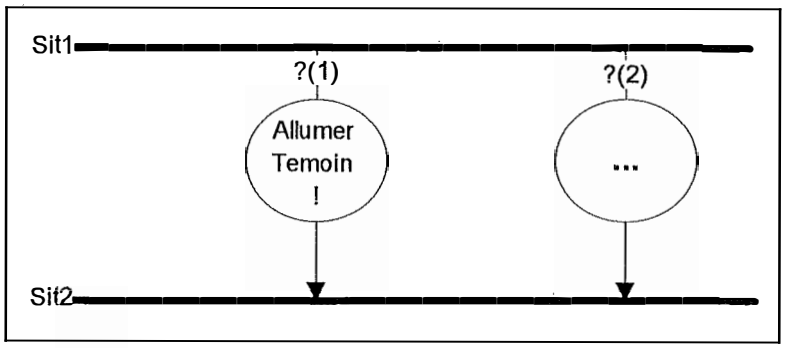
Style Etat Transition.

F(Produire | BacEntreeVide ∨ BacSortiePlein)
F(Rejeter | BacEntreeVide ∨ BacSortiePlein)



?(1) : Not BacEntreeVide and
Not BacSortiePlein

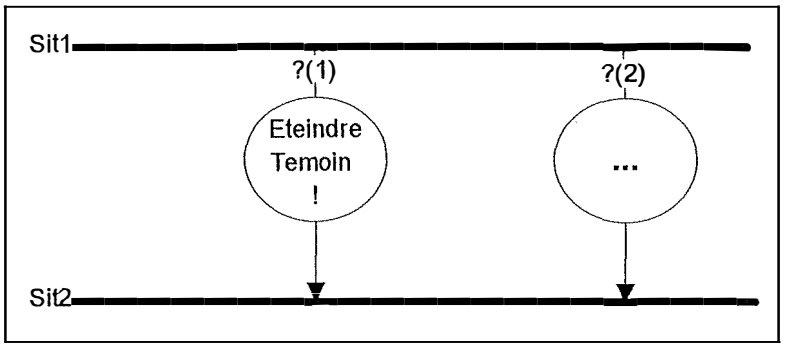
XO(AllumerTemoin | ⌉ Temoin ∧ BacSortiePlein)



?(1) :
not Temoin and
BacSortie

?(2) :
Condition
Complémentaire

XO(EteindreTemoin | Temoin ∧ ⌉ BacSortiePlein)

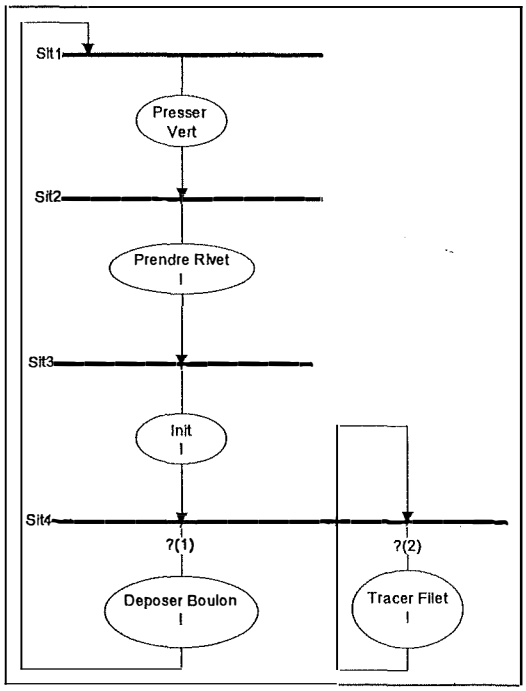


?(1) :
Temoin and not BacSortie

?(2) :
Condition Complémentaire

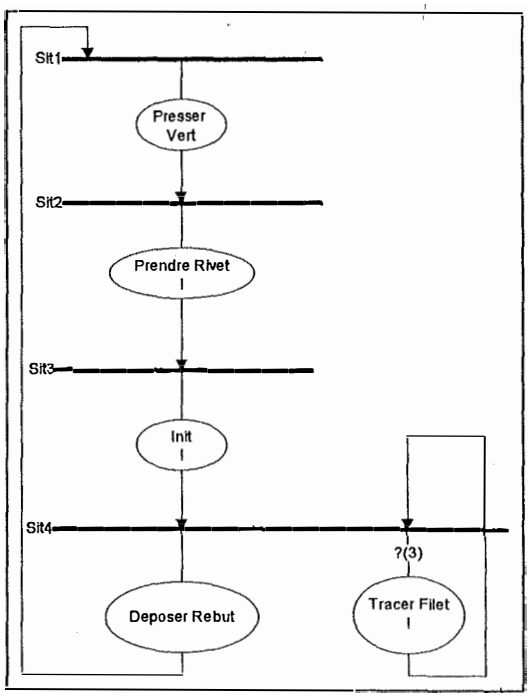
Style Action Composition

Produire ↔ Ouvrier.PresserVert ; PrendreRivet ; TracerTousFilets ;
DeposerBoulon
TracerTousFilets ↔ {TracerFilet}^{120}

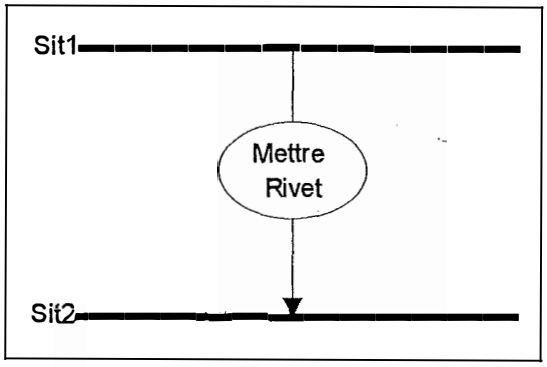


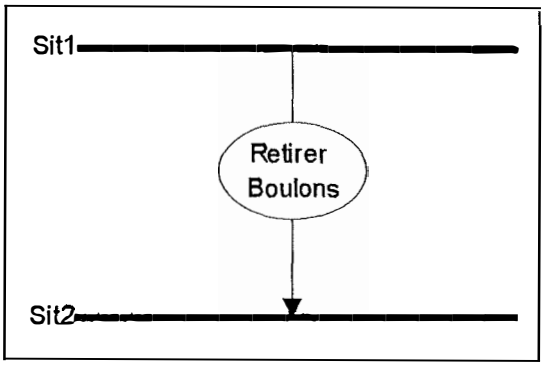
?(1) : NbrFilets > 120
?(2) : NbrFilets <= 120

Rejeter ↔ Ouvrier.PresserVert ; PrendreRivet ;
TracerPartieFilets ;
Avorter
TracerPartieFilets ↔ {TracerFiletOuRien}^{120}
TracerFiletOuRien ↔ TracerFilet ⊕ DAC
Avorter ↔ Ouvrier.PresserRouge ⊗ DeposerRebut

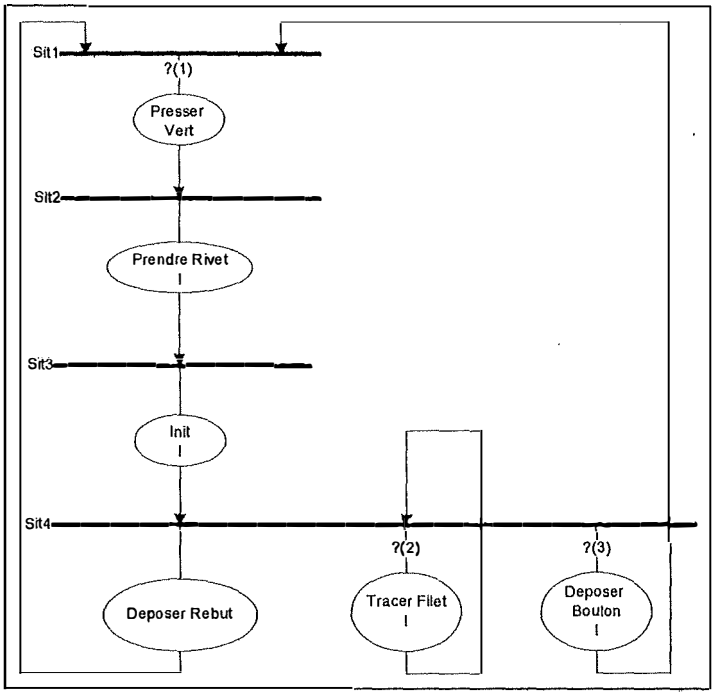


?(3) : NbrFilets <= 120





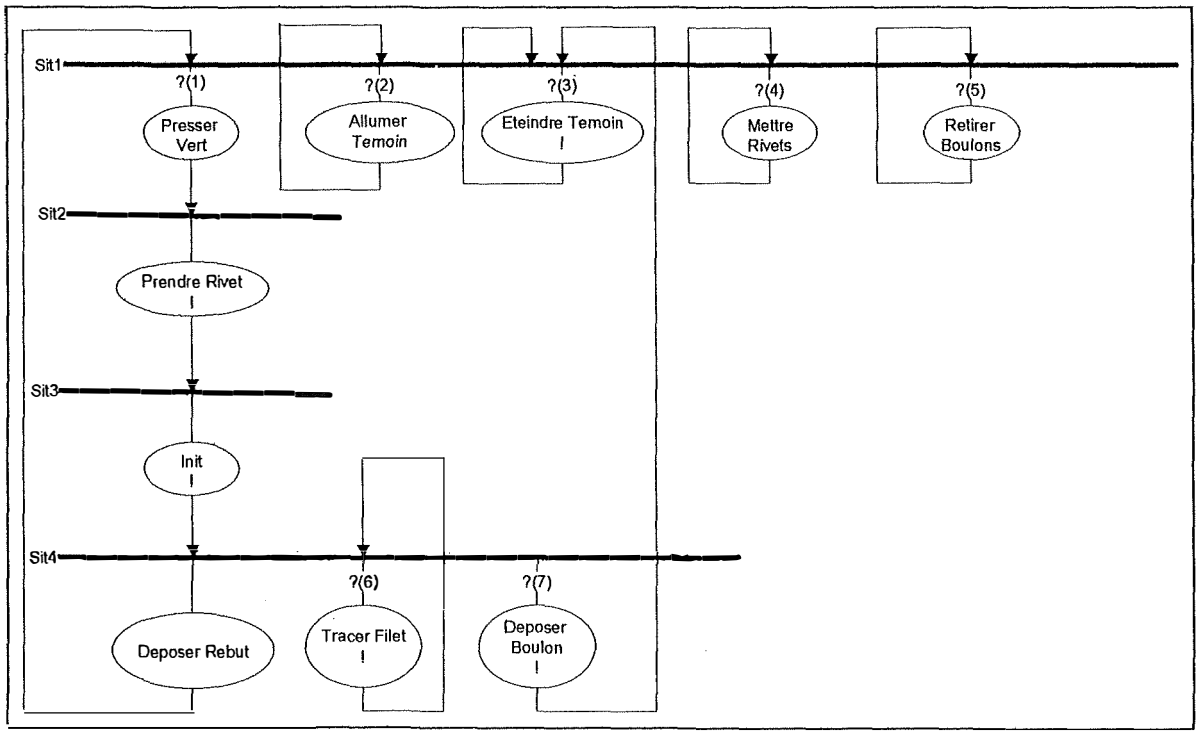
INTEGRATION LOCALE.




```
?(1): not
BacEntreeVide
 and not
BacSortiePlein
```

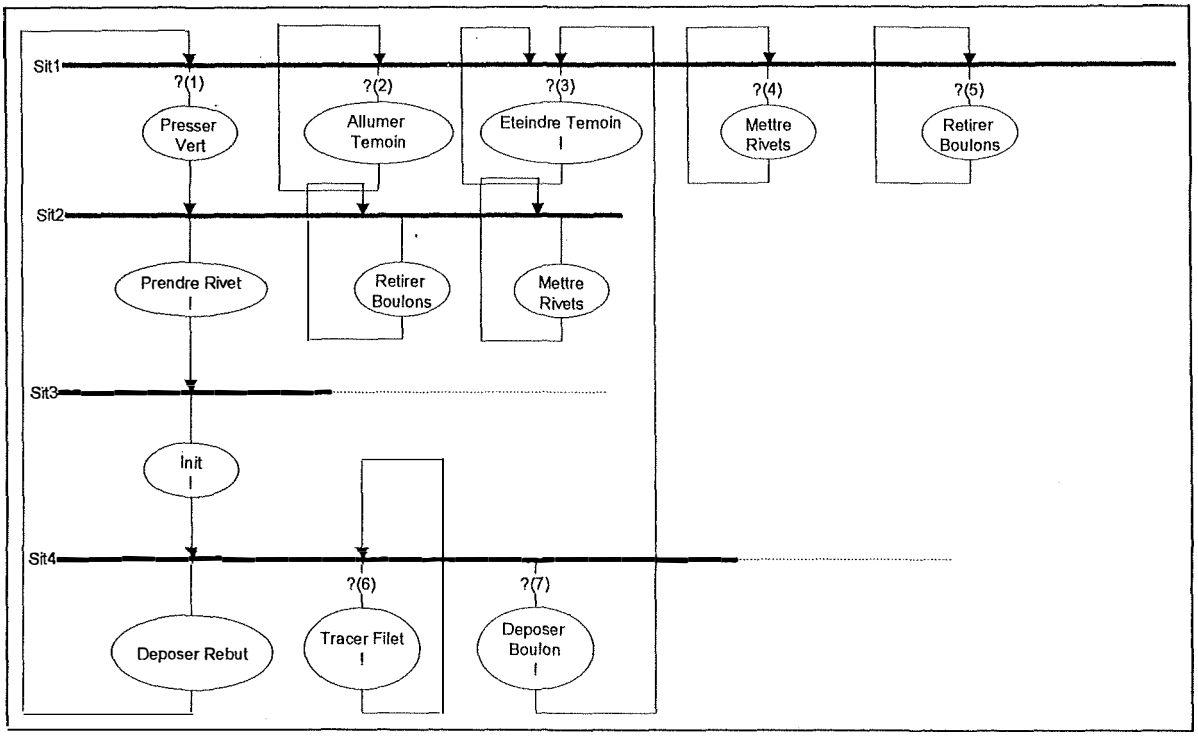
```
?(2): NbrFilets <= 120
?(3): NbrFilets > 120
```

INTEGRATION GLOBALE.




```
?(1) : not BacEntreeVide and not BacSortiePlein and not Temoin
?(2) : not Temoin and BacSortiePlein
?(3) : Temoin and not BacSortiePlein
?(4) : (Temoin and BacSortiePlein)
    or (not Temoin and not BacSortiePlein)
?(5) : (Temoin and BacSortiePlein)
    or (not Temoin and not BacSortiePlein)
?(6) : NbrFilets <= 120
?(7) : NbrFilets > 120
```

On remarque que les conditions se trouvant sur `sit1` ont changé, ceci est dû aux contraintes d'obligation exclusive des actions `AllumerTemoin` et `EteindreTemoin`.



Il faut remplacer les traits pointillés par des transitions dont les actions sont `RetirerBoulons` et `MettreRivets`.

CREATION DES APPELS.

Les seuls appels devant être créés sont les appels de naissance et de mort depuis l'objet OBSERVATEUR vers l'objet MACHINE.

TRADUCTION DES CONTRAINTES DE COOPERATION.

Dans cet exemple, étant donné qu'il n'y a qu'un seul objet, il est difficile de traduire des contraintes de coopération.

TRADUCTION DES « DERIVED COMPONENTS ».

Lors de la mise à jour des attributs, il faut ajouter partout où apparaît un des composants de la relation définissant les attributs dérivés, cette relation.

```
Operation PrendreRivet
     Updates
     NbrRivets        := NbrRivets - 1
     BacEntreeVide    := NbrRivets = 0
```

ERRATA.

- A la page 1.3, premier paragraphe, il faut lire « Ce formalisme est présenté sous la forme d'une structure dont la syntaxe et la sémantique sont une fois pour toute définies. » à la place de « Ce formalisme est présenté sous la forme d'une structure dont la syntaxe est une fois pour toute définie. ».
- A la page 1.6, section 1.2.6, premier paragraphe, il faut lire « ..., dont les contraintes sur les agents sont exprimées afin de définir un ensemble de vies admissibles. » à la place de « ..., dont les contraintes sur les agents sont exprimées afin de définir une ensemble de vies admissibles. ».
- A la page 1.9, section 1.3.2, troisième point de la liste, il faut lire « Le type construit PRENOMS est une séquence de STRING (Correspondant aux différents prénoms ordonnés d'une personne » à la place de « Le type construit PRENOMS est une séquence de STRING (Correspondant aux différents prénoms ordonnés d'un personne ».
- A la page 1.10, section 1.3.2, dernier paragraphe, il faut lire « ... puisque lors de cette définition un type portant le nom de l'agent est automatiquement associé. » à la place de « ... puisque lors de cette définition un type portant le nom de l'agent est automatique associé. ».
- A la page 1.13, dans l'exemple de la section 1.3.6.1.1, il faut lire « `Cab instance of CABINE` » à la place de « `Cabine instance of CABINE` ».
- A la page 1.14, section 1.3.6.2.1, fin du paragraphe, il faut lire « Le premier exemple exprime, pour l'agent ETAGE, le fait que si l'étage est égal à *PremierEtage*, alors le composant d'état *Appel_descente* a comme valeur *UNDEF.* » à la place de « Le premier exemple exprime, pour l'agent ETAGE, le fait que si et seulement si l'étage est égal à *PremierEtage*, alors le composant d'état *Appel_descente* a comme valeur *FALSE.* »
- A la page 1.16, section 1.3.6.2.3, fin du paragraphe, il faut lire « La seconde assertion exprime l'interdiction de l'action *Fermer_porte* si celle-ci est dans un état fermé ... » à la place de « La seconde assertion exprime l'interdiction de l'action *Fermer_porte* si celle-ci est dans un état ouvert ... ».
- A la page 2.4, section 2.2.3, il faut lire « Tout objet a une identité représentée par un « *surrogate* » et ...» à la place de « Tout objet a une identité par un « *surrogate* » et ... ».
- A la page 2.7, section 2.3.2, il faut lire « ..., les attributs hérités et les actions non modifiées se situent à la gauche du cercle représentant l'objet. » à la place de « les attributs et les actions héritées non modifiés se situent à la gauche du cercle représentant l'objet. ».

- A la page 2.11, section 2.3.4.2, il faut lire « L'éditeur est présenté à la Figure 2-8. » à la place de « L'éditeur vous est présenté à Figure 2-8. ».
- A la page 3.5, il manque les points 10 et 11, « Traduction des contraintes de coopération. », « Traduction des « Derived components ». ».
- A la page 3.10, section 3.2.9.1, dernier paragraphe, il faut lire « Donc, toutes les transitions ayant pour destination *PasDePanne* devront avoir pour effet de mettre l'attribut *Panne* à False. Dans ce cas, si un objet termine sa vie, et s'il est passé par la situation *Panne1* (où l'attribut *Panne* vaut True) alors il est passé après par la situation *PasDePanne* et ...» à la place de « Donc, toutes les transitions ayant pour destination PasDePanne devront avoir pour effet de mettre l'attribut *Panne1* à False. Dans ce cas, si un objet termine sa vie, et s'il est passé par la situation *Panne* (où l'attribut *Panne* vaut True) alors il est passé après par la situation PasDePanne et ...».
- A la page 3.13, section 3.2.9.2.3, dans l'exemple, il faut lire « `XO( Action | Cond )` » à la place de « `XO( Cond | Action )` ».
- A la page 3.22, section 3.2.9.3.3, dernière phrase, il faut lire « *Action2* hérite à la fois des « *Effect of Action* » de *Action1* et de *Action3*. » à la place de « *Action2* hérite à la fois des « *Effect of Action* » de *Action1* et de *Action2*. »

REMERCIEMENTS

Je tiens tout d'abord à remercier le Professeur Eric Dubois, promoteur de ce mémoire, pour son aide et ses éclaircissements lors de la réalisation de celui-ci, ainsi que Patrick Heymans pour ses nombreux conseils sur le langage ALBERT II.

Je remercie aussi Messieurs Pedro Inacio, Joao Faustino, Joao Gouveia et Gonçalo Silva ainsi que Madame Anna Barros qui m'ont permis durant 4 mois de m'intégrer dans l'équipe de développement d'OBLOG Software à Lisbonne. Ce stage a été des plus intéressants tant au niveau des connaissances qu'au niveau humain.

Je remercie enfin Bénédicte pour sa patience et le temps qu'elle a consacré à la relecture et à la correction de ce mémoire ainsi que ma mère pour son soutien tout au long de mes études.

RÉSUMÉ

L'ingénierie des besoins est actuellement considérée comme une activité critique dans le cadre du développement d'un logiciel. Nombre de langages formels ont été développés pour exprimer les besoins du client lors de cette phase. Nous nous intéresserons à l'un d'entre eux : ALBERT II qui s'intéresse plus particulièrement aux besoins des systèmes composites temps-réel. Notre objectif est de développer une méthodologie pour aider à la construction d'un prototype à partir d'une spécification en ALBERT II, le prototype devant servir l'analyste à valider la spécification. Nous porterons également notre attention sur le langage OBLOG qui a été utilisé pour réaliser ce prototype. Ce langage a comme caractéristique de pouvoir produire des programmes exécutables en un laps de temps très réduit. En effet, il a été développé pour la phase de conception (ingénierie du logiciel) et génère automatiquement le code du programme. C'est pourquoi ce langage a été choisi pour réaliser le prototype. La méthode de prototypage qui a été développée a permis de mettre en évidence les différences et les similitudes entre ALBERT II et OBLOG. Ce mémoire présente également une étude de cas complète et évalue les langages utilisés.

ABSTRACT

At the present time, Requirements Engineering is considered as a critical activity in the context of Software Development. Many formal specification languages have been designed. We will focus on one of them : ALBERT II which was developed more particularly for capturing requirements inherent to composite real-time systems. Our goal is to develop a methodology for building a prototype from an ALBERT II specification. This prototype will help the analyst to validate the specification. We will focus on the OBLOG language which was used for designing this prototype. This language is characterised by its ability to produce executable programs in a short period of time. Actually, it has been built for the design phase (Software Engineering) and it generates automatically source code. This is why we have chosen this language for prototyping. The prototyping method which have been developed helped to highlight the differences and the similarities between ALBERT II and OBLOG. In addition, this thesis shows a complete case study and evaluates the languages involved.

TABLE DES MATIERES.

TABLE DES ILLUSTRATIONS.

INTRODUCTION.

MOTIVATIONS.

Il est reconnu aujourd'hui que la phase la plus critique et la plus difficile du cycle de développement d'un logiciel est la phase d'analyse des besoins, au cours de laquelle l'informaticien décrit avec précision le système à mettre en place, ses objectifs, ses fonctions, etc. ...

C'est la phase la plus critique dans la mesure où elle est la première et que, à ce titre , elle conditionne toutes les phases ultérieures. Une erreur à ce niveau a toujours de lourdes répercutions tant au niveau des coûts qu'au niveau des efforts à fournir pour la corriger. Ces efforts et ces coûts seront d'autant plus importants que l'erreur est détectée tardivement dans le développement.

C'est aussi la phase la plus difficile parce que l'informaticien doit rédiger un cahier des charges à partir d'informations collectées auprès de clients et qui sont bien souvent imprécises, incomplètes voire même contradictoires. Ce travail difficile en soi nécessite une étroite collaboration entre l'informaticien et le client.

L'informatique accorde un intérêt croissant à l'analyse des besoins, au point d'en faire une discipline à part entière, appelée aussi ingénierie des besoins. Cette jeune discipline subit l'influence d'une part de l'apparition de langages formels de spécification des besoins et d'autre part de la prise de conscience que la spécification des besoins ne devait pas se limiter à décrire les systèmes informatiques mais également leur environnement.

C'est dans ce cadre que le langage ALBERT II a été développé. Ce langage est caractérisé par un aspect formel. Ce formalisme fournit un ensemble de règles rigoureuses d'interprétation qui garantissent la précision et l'absence d'ambiguïtés. De plus, ceci permet des raisonnements formels tels les contrôles de cohérence ou de complétude.

Malheureusement, le prix à payer pour l'utilisation de ce type de langage est un frein au niveau de la communication indispensable entre les informaticiens et les clients. Il faudra dès lors penser à un moyen pour faciliter cette communication. Plusieurs voies ont été explorées parmi lesquelles la génération automatique de documents en langage naturel à partir de spécifications formelles et le prototypage.

L'utilité du prototypage se situe au niveau de l'aide qu'il fournit pour valider les spécifications des besoins auprès des clients. Un prototype se caractérise par sa rapidité de développement et son faible coût. Pour réaliser ce prototype différents langages conviennent. Nous avons choisi l'un d'entre eux, OBLOG. Ce langage a été développé pour couvrir les

phases de conception et d'implémentation du cycle de vie d'un logiciel. Sa richesse se situe au niveau de l'implémentation puisque celle-ci est automatique.

Dans un premier temps, nous proposons dans ce mémoire de développer des règles de traduction d'une spécification en ALBERT II de cahiers des charges vers une spécification en OBLOG et dans un deuxième temps, nous critiquerons ces deux langages.

PLAN DE TRAVAIL.

Ce mémoire est divisé en trois parties, consacrées respectivement au langage ALBERT II, au langage OBLOG et à l'étude de prototypage.

ALBERT II.

Le chapitre 1 présente le contexte dans lequel ALBERT II à été développé, situe ce langage et introduit les outils. Les chapitres 2 et 3 définissent les divers concepts du langage ainsi que la structure d'une spécification. Dans les chapitres 4 et 5, nous développons une étude de cas basée sur un système d'ascenseur et nous la commentons. Les chapitres 6 et 7 présentent quelques commentaires sur le langage et l'éditeur.

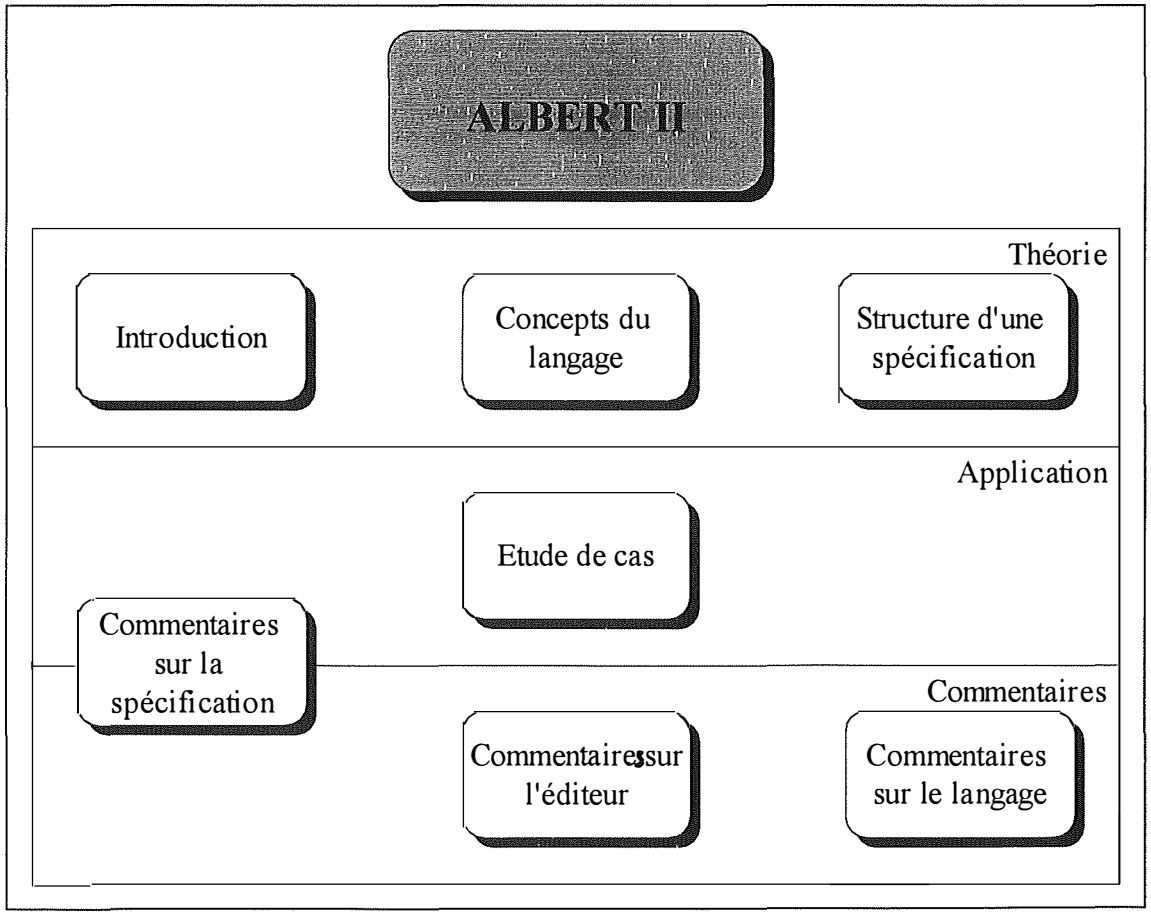


Figure 1 : Organisation des chapitres de la partie concernant ALBERT II.

OBLOG.

Le chapitre 1 présente le contexte dans lequel OBLOG à été développé et situe ce langage. Les chapitres 2, 3, 4 et 5 présentent les concepts du langage, l'outil CASE, ses mécanismes de fonctionnement et le langage utilisé pour rédiger les expressions. Le chapitre 6 commente la spécification présentée en annexe et le chapitre 7 le langage.

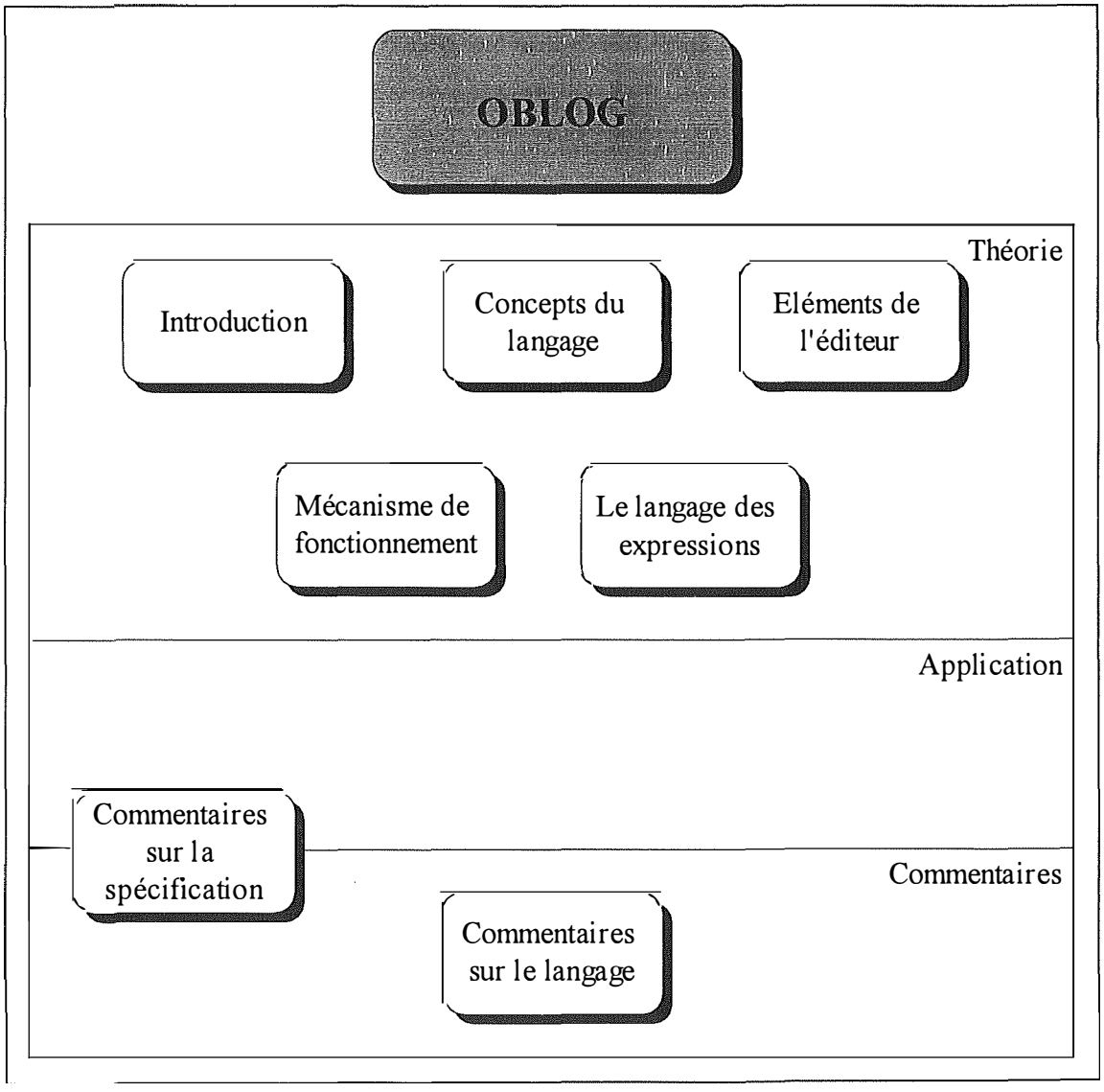


Figure 2 : Organisation des chapitres de la partie concernant OBLOG.

ETUDE DE PROTOTYPAGE.

Le chapitre 1 introduit ce qu'est un prototype et son utilité. Le chapitre 2 présente en détail la méthodologie qui a été développée pour générer des prototypes en OBLOG à partir de spécifications de cahiers des charges en ALBERT II. Le chapitre 3 propose un exemple d'application de cette méthodologie.

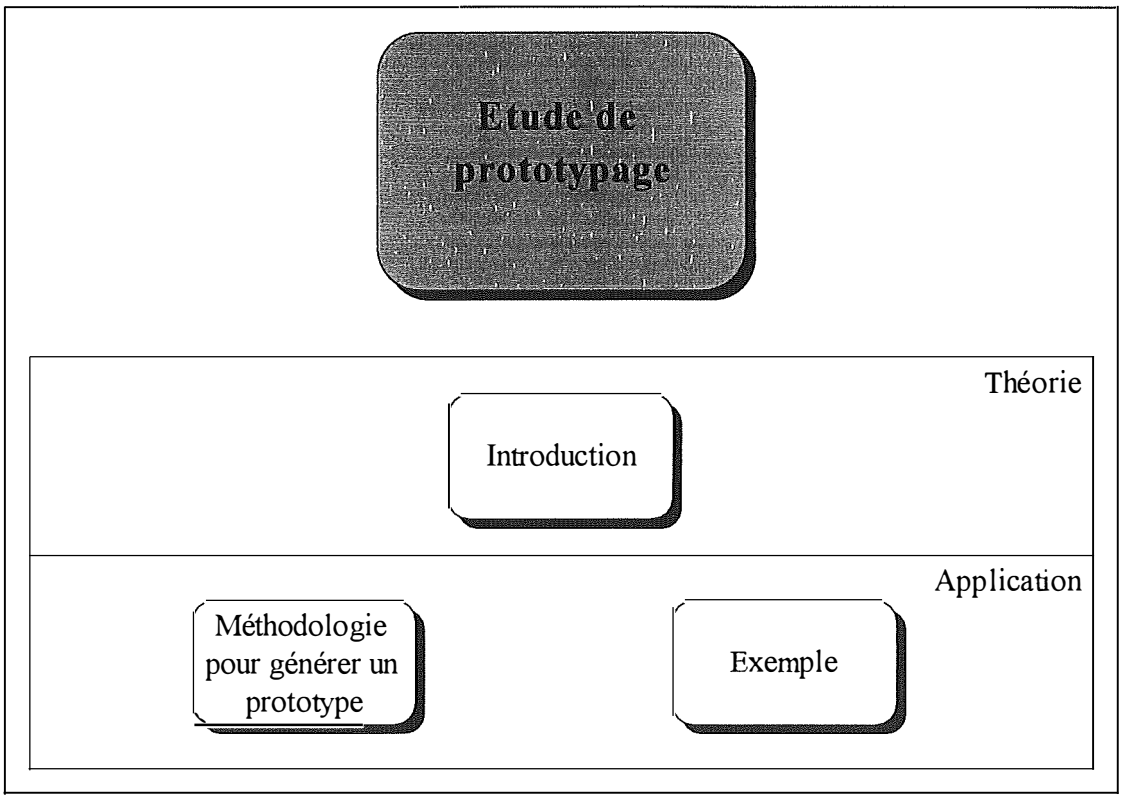


Figure 3 : Organisation des chapitres de la partie concernant la méthodologie.

1. ALBERT II.

1.1 INTRODUCTION.

Tout d'abord, il est important de situer un langage de spécification, de définir ce pour quoi il a été développé. ALBERT II [DUB95][JUN96], qui est l'acronyme de « *Agent-oriented Language For Building And Eliciting Requirements For Real Time Systems* », a été développé dans le cadre du projet ESPRIT : ICARUS [ICA95] et ce pour spécifier les besoins (« *Requirements* ») inhérents aux comportements des systèmes composites distribués complexes avec des contraintes de temps réel. Certains domaines comme le CIM (« *Computer Integrated Manufacturing* »), les applications en télécommunication et les systèmes inter ou intra organisation ont déjà permis de valider le langage.

1.1.1 LE LANGAGE.

Le langage ALBERT II se base sur une logique du premier ordre étendue par une logique temporelle et intégrant des aspects temps-réel.

Il est destiné à la phase d'ingénierie des besoins c'est-à-dire qu'il sert à exprimer les besoins du client dans le cahier des charges.

De ce fait, le langage doit posséder une grande expressivité afin de décrire non seulement les informations concernant le système informatique, leur traitement et les diverses contraintes s'y appliquant (entre autre les contraintes temporelles) mais également son environnement (à savoir le comportement des personnes, des machines, etc., ...).

Une autre caractéristique importante d'ALBERT II est le fait qu'il permette de faire le rapprochement de façon assez simple entre les concepts informels exprimés en langage naturel et ces mêmes concepts exprimés en ALBERT II. L'importance de cette caractéristique ressort lors des phases de vérification et de validation des documents formels de l'ingénierie des besoins. Ces documents seront d'autant plus facilement compris par le client qu'ils ne comportent pas d'artefacts (ce qui est souvent le cas avec les langages de spécification propre à l'ingénierie du logiciel). Les artefacts sont des éléments de la spécification n'ayant aucun correspondant dans l'univers du discours et qui ne sont présents que pour servir à des mécanismes propres au langage.

A cela vient s'ajouter l'impératif d'interprétation unique et non ambiguë des concepts exprimés. Ceci est motivé par le fait que le cahier des charges qui est présenté au client doit rencontrer au mieux ses besoins et ce de façon non ambiguë sans quoi, il est impossible de le valider. Aucun client n'acceptera un cahier des charges où certains concepts ne seraient pas

énoncés clairement et où l'interprétation de ces concepts porterait à confusion. Cet impératif n'est réalisé que par un certain degré de formalisme dans le langage utilisé. Ce formalisme est présenté sous la forme d'une structure dont la syntaxe est une fois pour toute définie. Cette syntaxe et cette sémantique permettront d'exprimer toutes les combinaisons permises entre les concepts propres au langage ainsi que de définir leur représentation.

Il est aussi à noter que le langage ALBERT II est non déterministe c'est-à-dire que le résultat de l'« exécution » d'une spécification peut donner des modèles valides mais pouvant varier d'une fois à l'autre.

La raison première du caractère orienté agent d'ALBERT II est liée au fait que ce langage a été développé pour les systèmes composites. Nous savons que ces systèmes sont caractérisés par une propriété globale qui doit être réalisée par le comportement coopératif des acteurs du système. Or, comme le concept d'agent s'accompagne de responsabilités contractuelles, il était normal de baser le langage sur ce concept. On a aussi remarqué, entre autre dans les domaines cibles d'ALBERT II, que le volume des spécifications étaient de plus en plus important. Pour faciliter la manipulation et la gestion de ces informations, il était nécessaire de structurer cette masse de données. Cette raison vient renforcer le choix du concept d'agent.

1.1.2 LES OUTILS

Autour de ce langage, les concepteurs d'ALBERT II ont développé une série d'outils tel un éditeur sous Windows 95 qui vous sera présenté et sur lequel nous ferons quelques commentaires, un outil de mise en page de spécification générant un texte en Latex à partir d'une spécification en Syntaxe ASCII et un prototype d'animateur [DDD94].

1.2 CONCEPTS DU LANGAGE.

1.2.1 AGENT.

Le concept d'agent se caractérise par l'ensemble des responsabilités qu'a l'agent en ce qui concerne les informations qu'il gère ou consulte et ses relations avec les autres agents. Les agents sont vus comme une spécialisation du concept d'objet. Les relations entre les agents se basent sur le concept de visibilité, qui est un lien statique entre des agents permettant de faire connaître les composants d'état et les actions entre ceux-ci. S'ajoutent à ce concept, les concepts de perception et d'information qui sont la partie dynamique de cette visibilité.

Pour bien cerner la différence entre un agent et un objet, voyons un exemple. Prenons une bibliothèque : dans cette bibliothèque, il y a des bibliothécaires qui prêtent des livres à des emprunteurs. En utilisant les concepts OO (Orienté Objet), nous aurons trois classes d'objets : la classe des bibliothécaires, la classe des utilisateurs et la classe des livres. Par contre, si nous nous orientons vers une conception OA (Orienté Agent), nous n'aurons plus que deux classes d'agents : les bibliothécaires et les emprunteurs. Dans cette deuxième approche, les livres ne sont pas des agents puisqu'ils n'ont au sein de ce système aucune responsabilité, ce qui n'est pas le cas des bibliothécaires et des emprunteurs.

1.2.2 SOCIETE.

Le concept de société se définit comme un agent complexe composé de plusieurs agents simples. Ceci permet une certaine hiérarchisation des agents en les regroupant dans diverses sociétés. Cette hiérarchie s'étend également aux sociétés elles-mêmes. La Figure 1-1 présente un exemple de cette hiérarchie.

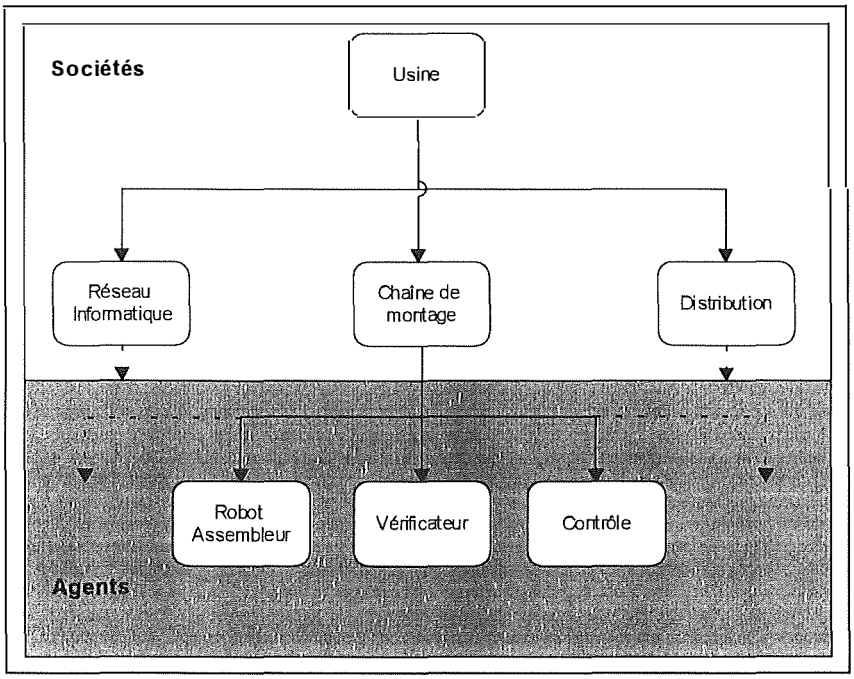


Figure 1-1 : Sociétés et agents

1.2.3 ACTION.

Dans la terminologie d'ALBERT II, une action dénote soit l'occurrence d'un événement ayant des effets directs sur l'état de l'agent soit l'occurrence d'un événement n'ayant pas d'effet direct. Les actions sont caractérisées par une durée.

Revenons à notre exemple de bibliothèque : Un emprunteur peut emprunter un livre auprès d'un bibliothécaire. Ceci est une action ayant un effet sur l'état de l'emprunteur puisque maintenant il est en possession d'un livre. Il est également possible aux bibliothécaires d'envoyer aux emprunteurs un rappel. Ceci est un exemple du second type d'action.

1.2.4 ETAT.

L'état d'un agent permet de connaître les caractéristiques de cet agent à un moment donné de sa vie . La valeur de cet état à un instant donné est la conséquence directe de la suite d'actions survenues dans la vie de l'agent jusqu'à cet instant.

1.2.5 VIE.

La vie d'un agent est la séquence alternée d'actions et d'états. La vie admissible d'un agent est la suite alternée d'actions et d'états vérifiant toutes les contraintes du système au cours de la vie. Une suite d'actions faisant partie de cette vie admissible s'appelle l'histoire.

Alors que la suite d'états s'appelle la trace. La Figure 1-2 donne un exemple de vie possible pour un agent.

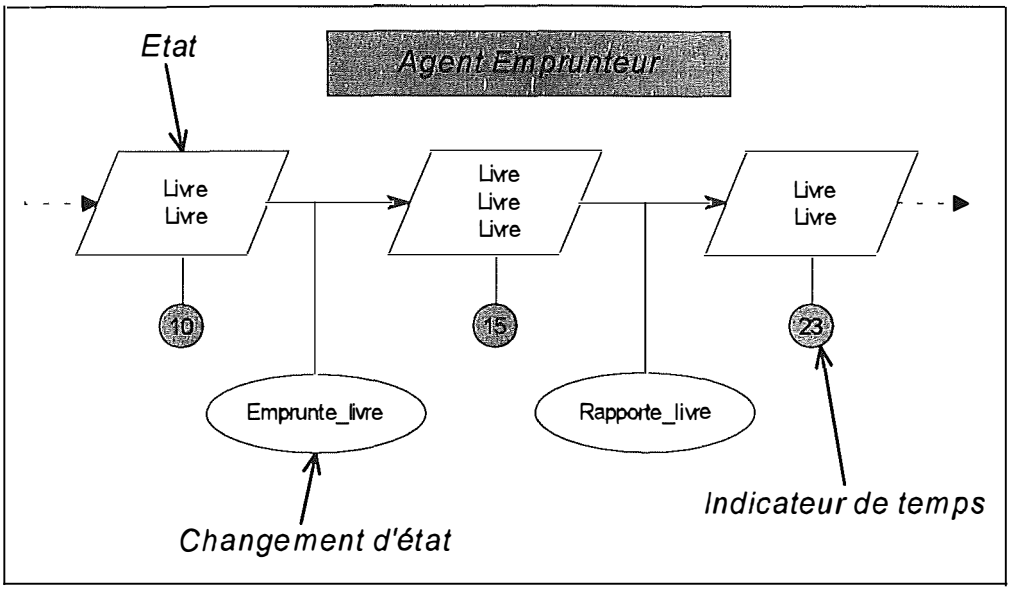


Figure 1-2 : Exemple de vie possible d'un agent

1.2.6 CONTRAINTES LOCALES.

Les déclarations définissent l'ensemble des vies possibles d'un agent. Pour restreindre l'ensemble de vies qui est généralement infini, des contraintes sur les agents sont exprimées afin de définir une ensemble de vies admissibles.

Il existe différents types de contraintes qui vont réduire le nombre de vies possibles des agents . Ces contraintes consistent en :

1. Une ou plusieurs relations qui doivent être vérifiées pour tous les états de l'agent (Figure 1-3.a).
2. Une ou plusieurs relations entre deux états d'un agent (Figure 1-3.b).
3. Une ou plusieurs relations entre un état et le début d'une action (Figure 1-3.c).
4. Une ou plusieurs relations entre la fin d'une action et l'état qui suit cette action (Figure 1-3.d).
5. Une ou plusieurs relations entre deux actions (Figure 1-3.e).

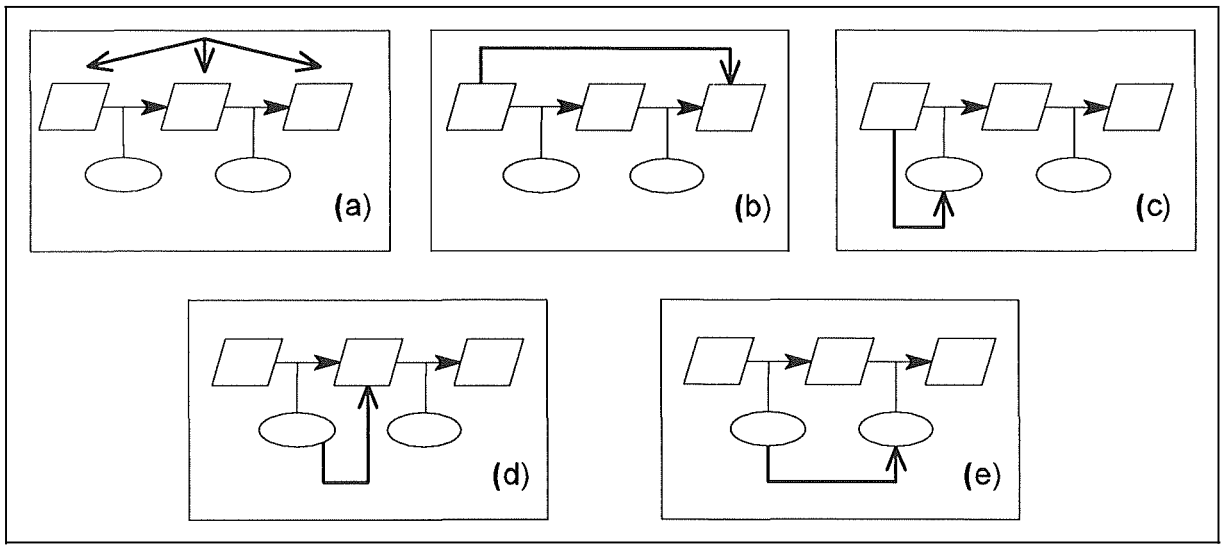


Figure 1-3 : Types de contraintes

1.2.7 CONTRAINTES DE COOPERATION.

Comme pour d'autres approches orientées objet, lorsqu'un agent veut faire connaître ses actions ou ses composants d'états aux autres agents, il doit les rendre visibles c'est-à-dire les exporter. C'est ce qu'on appelle la visibilité. L'exportation est une propriété statique des agents c'est-à-dire qu'elle n'évolue pas dans le temps. L'exportation se définit comme la déclaration de visibilité d'un composant d'état ou d'une action par un agent et elle implique automatiquement une importation de l'agent vers lequel on exporte.

Les concepts de perception et d'information viennent enrichir le concept de visibilité. Pour bien les comprendre, voyons l'exemple d'un dialogue entre 2 personnes (Figure 1.4). L'émetteur possède un certain nombre d'informations qu'il désire ou non partager avec son interlocuteur (premier type de contraintes). Le récepteur est prêt ou non à percevoir ces informations (deuxième type de contraintes) et ces états peuvent évoluer dans le temps. Nous avons donc un caractère dynamique pour ces propriétés.

Maintenant, ces concepts restent valables pour les composants d'état (le flux de données) et les actions (le flux de contrôle).

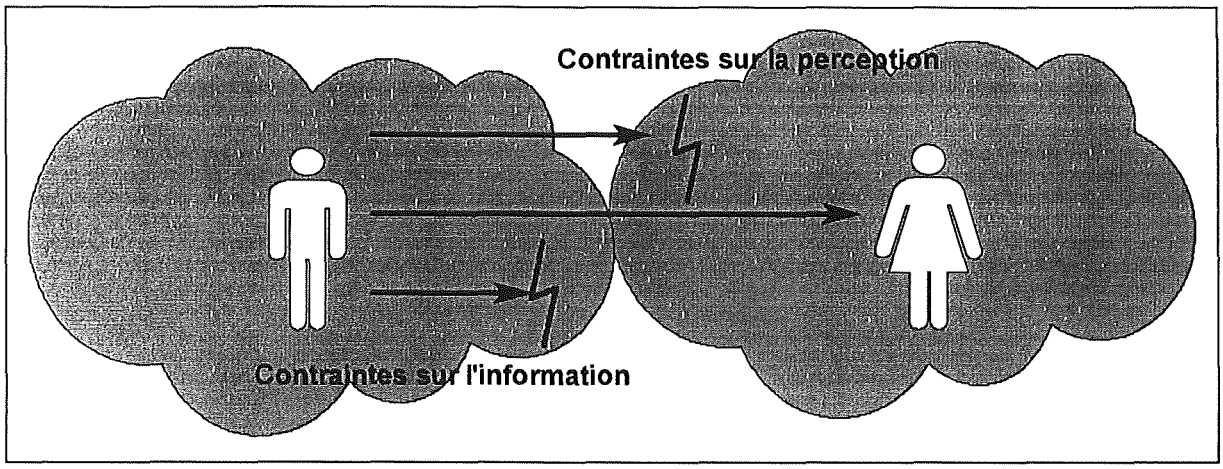


Figure 1-4 : Les contraintes d'information et de perception.

1.3 STRUCTURE D'UNE SPECIFICATION.

1.3.1 TYPES PREDEFINIS.

Il existe 5 types de données prédéfinis qui sont les suivants :Booléen, Entier, Rationnel, Caractère et chaîne de caractères (string). De plus, pour chacun de ces types, les opérations habituelles ont été définies. Nous vous renvoyons au chapitre 3 du manuel de référence d'ALBERT II pour de plus amples informations.[JUN96]

1.3.2 TYPES DEFINIS PAR L'UTILISATEUR.

Il est possible en ALBERT II de définir soi-même ses propres types de données grâce à une série de constructeurs. Ces constructeurs sont présentés dans le Tableau 1-1.

Types construits	Exemples
CP[T_1, T_2, ..., T_n]	PERSONNE = CP[Nom : STRING, Prénoms : PRENOMS, Age : AGE]
SET[T]	CLIENTS = SET[PERSONNE]
SEQ[T]	PRENOMS = SEQ[STRING]
BAG[T]	LISTE_COURSE = BAG[CP[Produit : STRING, Prix : RATIONAL]]
TABLE[T_1, T_2]	PROMOTION = TABLE[Client : PERSONNE, Réduction : RATIONAL]
UNION[T_1, T_2, ..., T_n]	CLIENTS_POTENTIELS = UNION[CLIENTS, CLIENTS_DU_CONCURRENT]
T*	AGE = INTEGER*
ENUM[s_1, s_2, ..., s_n]	TYPE_CLIENT = ENUM[GROSSISTE, PARTICULIER]

Tableau 1-1: Constructeurs de type avec exemples

Dans ce tableau, nous donnons quelques exemples de types construits que l'on explique comme suit :

1. Les caractéristiques d'une personne qui nous intéressent sont son nom, ses prénoms et son âge. Le type qui y correspond le mieux est un produit cartésien d'une STRING, d'un type construit PRENOMS et d'un type construit AGE.
2. L'ensemble CLIENT est un ensemble de PERSONNE.
3. Le type construit PRENOMS est une séquence de STRING (Correspondant aux différents prénoms ordonnés d'un personne).
4. La liste des courses est un ensemble pouvant avoir plusieurs éléments identiques (ce qu'on appelle un bag ou multi-ensemble). De plus à chaque produit correspond un prix.
5. La structure de table permet d'associer à une entrée, dans notre cas une personne, une valeur d'un certain type, ici une réduction.

6. L'union est l'ensemble construit par l'union des différents ensembles sur lesquels porte l'opération.
7. Le type AGE* est équivalent à l'union de AGE et {UNDEF}.
8. ENUM est l'opérateur permettant de construire un ensemble énuméré.

Au point 7, nous introduisons la constante UNDEF qui est la représentation de l'absence de valeur.

Il est également possible de restreindre l'ensemble défini par ces constructeurs grâce à une formule logique. L'exemple suivant vous présente la syntaxe d'une telle expression :

```
CLIENTS_INTERESSANTS = SET[PERSONNE] with ∀ p : Age(p) <> UNDEF ⇒
Age(p) > 35.
```

A cela vient s'ajouter les types définis implicitement lors de la définition des agents puisque lors de cette définition un type portant le nom de l'agent est automatique associé.

1.3.3 OPERATIONS DEFINIES PAR L'UTILISATEUR.

Il est également possible de définir des opérations sur les différents types construits. Ces opérations sont des fonctions mathématiques enrichies par des contraintes exprimées grâce à la logique du premier ordre, comme le montrent les 2 exemples suivants :

```
Choisir_un_client : CLIENTS → PERSONNE
Choisir_un_client(c) = p
with p ∈ c

Vieillir : PERSONNE → PERSONNE
Vieillir(p) = p'
with (Nom(p) = Nom(p'))
and  (Prénom(p) = Prénom(p'))
and  (Age(p) = UNDEF ⇒ Age(p') = UNDEF)
and  (Age(p) <> UNDEF ⇒ Age(p)+1 = Age(p'))
```

1.3.4 DECLARATION DES SOCIETES.

Chaque société doit être déclarée. Dans cette déclaration, on indique tous les agents faisant partie de cette société et si ces agents font partie d'une classe avec une ou plusieurs instances. La Figure 1-5 montre comment on représente graphiquement cette déclaration. Dans cette figure, l'AGENT1 est une classe dont le nombre d'instances est un et l'AGENT2 est une classe dont le nombre d'instances est illimité. La flèche entre l'AGENT1 et l'AGENT2 indique qu'il y a une exportation de soit un composant d'état, soit une action depuis l'AGENT1 vers l'AGENT2.

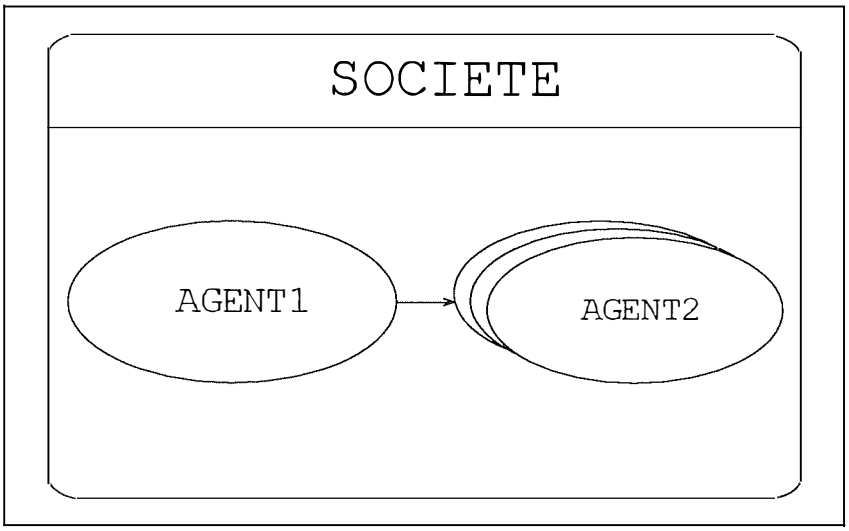


Figure 1-5 : Déclaration d'une société.

1.3.5 DECLARATION DES AGENTS.

1.3.5.1 Composants d'état.

La structure qui a été mise en place dans ALBERT II pour représenter la valeur de l'état s'appelle les composants d'état. Ces composants d'état sont typés et peuvent être individuels ou représenter une population. Ces populations peuvent être des ensembles, des séquences ou des tables. Les représentations graphiques des composants d'états sont présentées à la Figure 1-6 . La Figure 1-6 (a) représente un composant d'état individuel pouvant varier dans le temps, la Figure 1-6 (b) un composant d'état individuel constant, la Figure 1-6 (c) un ensemble, la Figure 1-6 (d) une séquence et la Figure 1-6 (e) une table. De plus, les composants d'états des Figure 1-6 (c) et (e) sont respectivement exporté vers un agent et importé depuis un agent. La flèche liant la Figure 1-6 (a) et la Figure 1-6 (d) a pour signification que le premier (a) est un composant dérivé et que la relation le définissant se base sur (d).

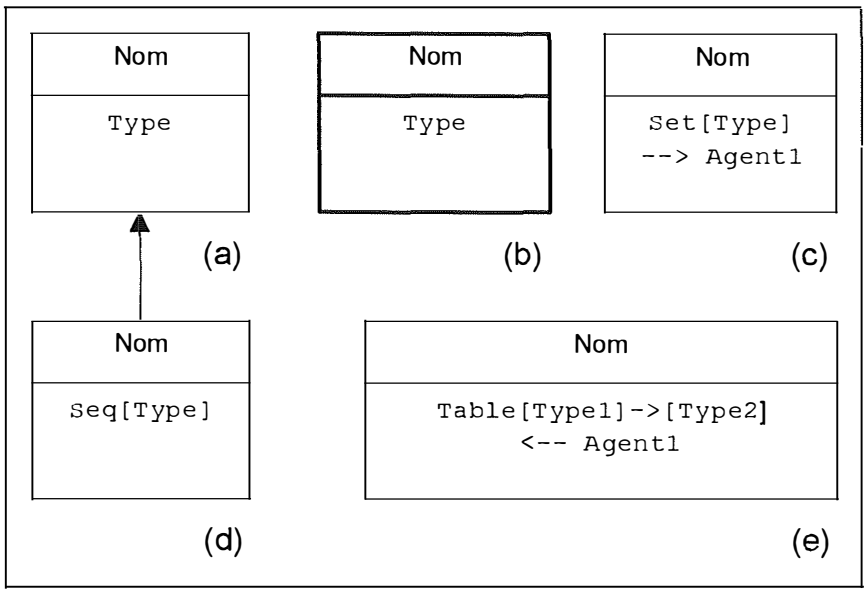


Figure 1-6 : Représentations graphiques des composants d'état.

1.3.5.2 Actions de l'agent.

En plus de ses composants d'état, un agent est caractérisé par des actions. La Figure 1-7 présente les différentes représentations graphiques des actions. La Figure 1-7 (a) correspond à une action sans paramètre, alors que la Figure 1-7 (b) correspond à une action avec deux paramètres, un de type `Integer` et l'autre de type `Char` . La Figure 1-7 (c) est la représentation d'une action qui fait obligatoirement partie d'une « *action composition* » c'est-à-dire qu'elle fait partie d'une action plus globale et qu'elle ne peut se réaliser seule. Les Figure 1-7 (d et e) indiquent qu'il y a respectivement une exportation de cette action vers un agent et une importation de cette action depuis un agent.

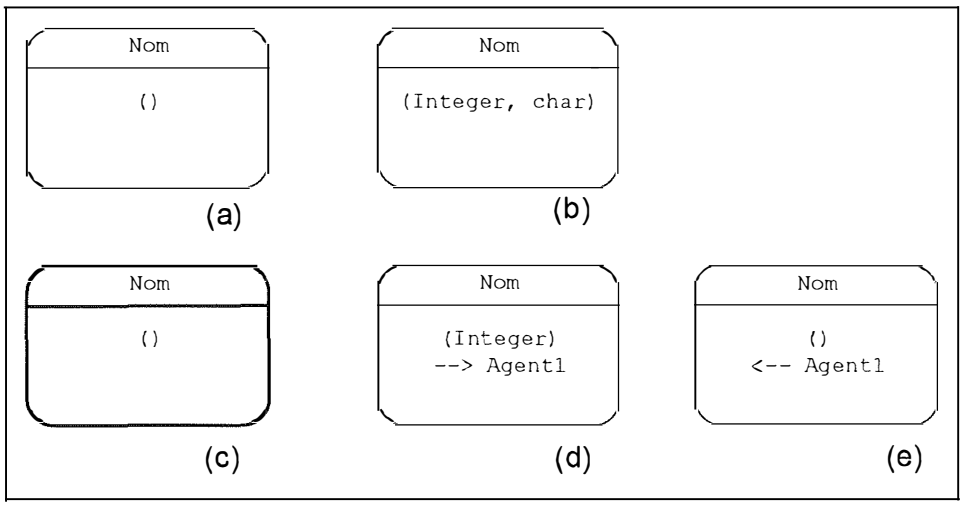


Figure 1-7 : Représentations graphiques des actions

1.3.6 CONTRAINTES SUR LES AGENTS.

Les contraintes sont classées en trois familles : « *Basic Constraints* », « *Local Constraints* » et « *Co-operation Constraints* ». Les « *Basic Constraints* » servent à décrire l'état initial d'un agent et les règles de dérivation des composants d'état dérivés. Tandis que les « *Local Constraints* » décrivent le comportement interne de l'agent. Et les « *Co-operation Constraints* » permettent de décrire les contraintes relatives à l'information et la perception des composants d'état et des actions de l'agent.

1.3.6.1 Basic Constraints.

Cette section est composée des sous-sections suivantes : « *Derived Components* » et « *Initial Valuation* » .

1.3.6.1.1 Derived components.

Les composants dérivés servent à décrire les dépendances statiques entre les composants d'état. Ces contraintes ont la forme de relations mathématiques entre un composant dérivé et d'autres composants d'état du même agent.

Dans l'exemple qui suit, l'assertion exprime la relation définissant le composant d'état *Appelmontetagesup* comme étant vrai s'il existe un ETAGE supérieur pour lequel il y a un appel de montée.

Exemple :

```
STATE COMPONENTS

Appelmontetagesup instance of BOOLEAN
Cabine instance of CABINE

BASIC CONSTRAINTS

DERIVED COMPONENTS

Appelmontetagesup ≡ ∃ E : Avant(Cab.Pos,E)∧ E.Appel_montee
```

1.3.6.1.2 Initial valuation.

Cette contrainte permet d'exprimer la valeur de l'état d'un agent pour le premier état d'une vie possible. Il est à remarquer que l'initialisation n'est pas obligatoire. Ceci a pour implication que le composant d'état pourra avoir n'importe quelle valeur pour le premier état de la vie, ensuite il évoluera en respectant les contraintes. Il est également évident que les

composants dérivés ne peuvent pas être initialisés. La valeur d'initialisation ne peut pas être une valeur calculée, elle doit être un littéral. L'exemple suivant initialise *Panne_a_venir* à vrai.

Exemple :

```
INITIAL VALUATION
Panne_a_venir = TRUE
```

1.3.6.2 Local Constraints.

La section « *Local Constraints* » est composée des sous-sections suivantes : « *State Behaviour* », « *Effects of Actions* », « *Capability* », « *Action Composition* » et « *Action Duration* ».

1.3.6.2.1 State behaviour.

Dans cette section, seules les contraintes concernant l'état sont exprimées (Figure 1-3 (a et b)). Il existe deux types de contraintes ne portant que sur les états. Les premières se rapportent aux propriétés mêmes d'un état sans tenir compte des autres états de la trace (ces contraintes portent aussi le nom d'invariant). Les deuxièmes expriment les liens qu'il y a entre deux états (on parle également de contraintes d'évolution). Un exemple d'invariant et de contrainte d'évolution sont présentés ci-dessous. Le premier exemple exprime, pour l'agent ETAGE, le fait que si et seulement si l'étage est égal à *PremierEtage*, alors le composant d'état *Appel_descente* a comme valeur *FALSE*. Le second exemple exprime le fait que si le composant d'état *Etat_porte* vaut TRUE, il devra passer par la valeur FALSE dans le futur.

Exemple :

```
STATE BEHAVIOUR
self = PremierEtage ⇒ Appel_descente = UNDEF
Etat_porte ⇒ ◊ ~Etat_porte
```

Remarque : SELF représente l'instance de l'agent dans lequel il est utilisé.

1.3.6.2.2 Effects of Actions.

Dans cette section, on déclare les effets des actions internes ou importées pour l'agent. Seules les actions ayant un effet y seront reprises. Il existe deux règles dont il va falloir tenir compte :

1. Un composant d'état ne changera jamais, s'il n'y a pas de « *effect of action* » appliqué sur lui.

2. Les actions ne peuvent être simultanées que s'il n'y a pas de « *effect of action* » conflictuel c'est-à-dire portant sur les mêmes composants d'état.

Exemple :

```
EFFECTS OF ACTIONS

Detecter_panne : Panne_a_venir := TRUE
Aller(dest) : destination := dest ; Etat_moteur := ENMARCHE
Controle.Bloquer_porte(Code_secret) : Etat_porte := FALSE
Vendre(_, quantite,_) with quantite <= Stock : Stock := Stock -
quantite
```

Il faut lire ces exemples comme suit et ce, à condition que les actions soient perçues :

- Lorsqu'il y a une occurrence de *Detecter_panne*, le composant d'état *Panne_a_venir* est mis à TRUE.
- Chaque fois qu'une occurrence de *Aller* a lieu, le composant d'état *destination* est mis à la valeur passée à l'action et le composant d'état *Etat_moteur* est mis à ENMARCHE.
- Lorsque l'action *Bloquer_porte* se produit dans l'agent *CONTROLE* avec comme valeur de paramètre *Code_secret*, le composant d'état *Etat_porte* est mis à FALSE.
- L'occurrence de l'action *Vendre* a pour effet de soustraire du composant d'état *Stock* la valeur *quantite* si celle-ci est plus petite que *Stock*. Les préconditions sur les paramètres des actions sont exprimées grâce à la clause *with*. Le signe « _ » a pour signification qu'il y a un paramètre mais que l'on ne s'intéresse pas à sa valeur dans l'expression.

On peut aussi remarquer que le « *effect of action* » est exprimé par une relation entre d'une part un composant d'état résultat et d'autre part les paramètres de l'action et les composants de l'état précédant le début de l'action (**Figure 1-3** (d)).

1.3.6.2.3 Capability.

Les contraintes exprimées dans cette section décrivent le lien entre la réalisation d'une condition portant sur l'état et l'occurrence d'une action (**Figure 1-3** (c)). Par défaut, une action **peut** avoir lieu. Il existe trois types de comportements introduits dans le langage :

1. L'**obligation** exprimant le fait qu'une action doit impérativement avoir lieu si la condition est vérifiée. (représentée par *O*)
2. L'**interdiction** exprimant le fait qu'une action ne peut se produire si la condition est vérifiée. (représentée par *F*)
3. L'**obligation exclusive** exprimant le fait qu'une action aura lieu si et seulement si la condition est vérifiée. (représentée par *XO*)

Exemple :

```
CAPABILITY
XO( Ouvrir_porte(Code) | Code_secret = Code )
F( Fermer_porte | ¬Etat_porte )
O( Commander(Produit) | Produit.Quantite < Produit.Seuil_Critique )
```

L'exemple précédent nous montre comment utiliser les « *Capability* » : La première assertion permet d'exprimer que l'occurrence de l'action *Ouvrir_porte* n'a lieu que si le *Code_secret* est bien égal au *Code* et uniquement dans ce cas-là. La seconde assertion exprime l'interdiction de l'action *Fermer_porte* si celle-ci est dans un état ouvert (représenté par le composant d'état *Etat_porte* à *False*). Enfin, la dernière assertion exprime l'obligation stricte de *Commander* un *Produit* si la *quantite* de celui-ci est inférieure au *Seuil_Critique*.

1.3.6.2.4 Action Composition.

Les contraintes d'« *Action Composition* » permettent de décrire comment une action interne est décomposée en d'autres actions internes ou importées. Il existe 5 types de combinaisons d'actions :

1. La séquence, représentée par « Act1 ; Act2 ; Act3 ; ... », qui signifie que les actions se suivent séquentiellement.
2. La répétition séquentielle, représentée par « $\{Act1\}^n$ », qui signifie que la même action se répète n fois séquentiellement.
3. La conjonction, représentée par « Act1 $\otimes$ Act2 $\otimes$ Act3 $\otimes$ ... », qui signifie que les actions se produisent simultanément (même début, même fin).
4. La conjonction parallèle, représentée par « Act1 || Act2 || Act3 || ... », qui signifie que toutes les actions se produisent et ce, sans aucune contrainte. Ce peut être une séquence, une conjonction, un mélange des deux ou un chevauchement.
5. La disjonction exclusive, représentée par « Act1 $\oplus$ Act2 $\oplus$ Act3 $\oplus$ ... », qui signifie que seule une des actions peut avoir lieu.

De plus, on a introduit dans le langage la « *Dummy Action* » symbolisée par DAC pour représenter la possible occurrence d'une action grâce à une disjonction exclusive combinant l'action potentielle et la « *Dummy Action* » (Act1 $\oplus$ DAC).

Il est également possible de contraindre les arguments des actions faisant partie d'une « *Action Composition* » grâce à la clause *with*.

L'occurrence des actions peut être limitée au cadre des « *Action Composition* ». Ceci a pour effet d'éviter que l'occurrence de l'action ainsi contrainte ne se produise en dehors du

cadre de la composition. Cette restriction peut être locale (L'action ne peut alors avoir lieu que dans des « *Action Composition* » de l'agent) ou extérieure (L'action ne peut se produire dans l'agent vers lequel elle est exportée que dans une « *Action Composition* »). Elle peut aussi présenter les deux caractéristiques à la fois.

Exemple :

```
ACTIONS

*Ouvrir_porte
*Attendre(sec)
*Fermer_porte
*Donner_bonbon
Ouvrir1
Ouvrir2

ACTION COMPOSITION

Ouvrir2 ↔ Ouvrir_porte ;Attendre(s) with s < 20 ;Fermer_porte
Ouvrir1 ↔
 Ouvrir_porte ;Attendre(s) with s >=
20 ;Donner_bonbon ;Fermer_porte
```

Cet exemple exprime dans un premier cas le fait que les actions *Ouvrir_porte*, *Attendre(s)* et *Fermer_porte* ne peuvent se produire que dans cet ordre si l'argument d'*Attendre* est inférieur à 20. Dans le deuxième cas, si l'argument d'*Attendre* est supérieur ou égal à 20, alors la séquence est la suivante : *Ouvrir_porte*, *Attendre(s)*, *Donner_bonbon* et *Fermer_porte*.

1.3.6.2.5 Action duration.

La section d'« *Action duration*» permet de contraindre la durée des actions internes. On a également introduit une clause *with* pour restreindre la portée des « *action duration* » aux actions qui vérifient la relation entre les arguments et les composants d'état.

Exemple :

```
ACTION DURATION

|Attendre_5s| = 5 s
|Ouvrir_porte| = 0 s
|Attendre(nbr_sec)| > nbr_sec with nbr_sec < Maximum_attente_sec
```

Nous pouvons tirer de cet exemple que la durée de *Attendre_5s* est de 5 secondes alors que l'action *Ouvrir_porte* est instantanée. La troisième assertion exprime le fait que l'action *Attendre* dure au moins *nbr_sec* si *nbr_sec* est inférieur à *Maximum_attente_sec*.

1.3.6.3 Co-operation Constraints.

La section « *Co-operation Constraints* » est composée des sous-sections suivantes : « *Action Perception* », « *Action Information* », « *State Perception* » et « *State Information* ».

1.3.6.3.1 Action Perception.

L'« *Action Perception* » permet de décrire la sensibilité d'un agent à l'occurrence d'actions extérieures. Il est bien clair que ces actions doivent avoir été exportées par les agents désirant interagir . Le langage utilise trois symboles K, I, XK pour définir la perception des actions. Par défaut, une action peut être perçue.

1. K exprime la connaissance de l'occurrence d'une action externe. Cette connaissance, pour autant que la condition soit vérifiée, a pour effet de modifier le comportement de l'agent concerné.
2. I exprime l'ignorance de l'occurrence d'une action externe. Dans ce cas, si la condition est vérifiée alors le comportement de l'agent concerné n'est en rien modifié lors de l'occurrence de l'action.
3. XK exprime la combinaison des deux expressions K(agent.action | condition) et I(agent.action | ¬ condition).

Cette contrainte s'accompagne également d'une clause *with* qui permet de réduire la non-perception des actions ou d'étendre la portée des expressions.

Exemple :

```
CO-OPERATION CONSTRAINTS

ACTION PERCEPTION

I( Utilisateur.Choisir_destination(dest) | Etages_arret[dest] )
K( Moteur.Descendre_cabine | TRUE )
XK( Utilisateur.Commander_ouverture | Moteur.Etat = ARRETE )
```

De cet exemple, nous pouvons dire que

- L'action *Choisir_destination* réalisée par l'UTILISATEUR est ignorée si l'entrée de *dest* dans la table *Etages_arret* est déjà à vrai.
- L'action *Descendre_cabine* commandée par le MOTEUR est toujours perçue.
- L'action *Commander_ouverture* réalisée par l'UTILISATEUR n'est perçue que si MOTEUR.*etat* vaut ARRETE.

1.3.6.3.2 State Perception.

Dans la section « *State Perception* », nous définissons comment un agent perçoit les composants d'états qui lui sont rendus visibles. Nous utilisons les mêmes symboles qu'à la section précédente. Par défaut, un composant d'état peut être perçu.

1. K exprime la connaissance du composant d'état externe. Cette connaissance est vraie pour autant que la condition soit vérifiée.
2. I exprime l'ignorance du composant d'état externe. Dans ce cas, si la condition est vérifiée alors l'agent concerné ignore la valeur de ce composant.
3. XK exprime la combinaison des deux expressions K(agent.composant | condition) et I(agent.composant | ¬ condition).

Exemple :

```
CO-OPERATION CONSTRAINTS
STATE PERCEPTION
K( Moteur.Etat | TRUE )
XK( _.Panne_a_venir | ¬Occupe )
```

De cet exemple, nous pouvons dire que

- A tout moment, l'agent connaît la valeur de *Etat* du MOTEUR.
- L'agent connaît la valeur de *Panne_a_venir* si et seulement si il n'est pas dans un état occupé.

1.3.6.3.3 Action Information.

Dans la section « *Action Information* », nous définissons sous quelles conditions un agent fait connaître l'occurrence des actions qu'il exporte. Nous utilisons les mêmes symboles qu'à la section précédente. Par défaut, l'occurrence d'une action peut être connue.

1. K conditionne la divulgation de l'information concernant l'occurrence de l'action. Cette divulgation a lieu pour autant que la condition soit vérifiée.
2. I conditionne le secret de l'occurrence de l'action. Dans ce cas, si la condition est vérifiée alors l'agent ne fait pas savoir qu'il réalise l'action.
3. XK exprime la combinaison des deux expressions K(action.agent | condition) et I(action.agent | ¬ condition).

Cette contrainte s'accompagne également d'une clause *with*.

Exemple :

```
CO-OPERATION CONSTRAINTS

ACTION INFORMATION

XK( Appeler( _ , Etage1 ).Etage2 | Etage1 = Etage2 )
K( Commander_ouverture.Cabine | TRUE )
```

De cet exemple, nous pouvons dire que

- L'action *Appeler* lorsqu'elle est réalisée ne peut être connue que par l'ETAGE qui est passé comme paramètre.
- L'agent fait toujours connaître l'occurrence de l'action *commander_ouverture* aux agents vers lesquels il exporte cette action.

1.3.6.3.4 State Information.

Dans la section « *State Information* », nous définissons sous quelles conditions un agent fait connaître les composants d'état qu'il exporte. Nous utilisons les mêmes symboles qu'à la section précédente. Par défaut, les composants d'état peuvent être connus.

1. K conditionne la divulgation des composants d'état. Cette divulgation a lieu pour autant que la condition soit vérifiée.
2. I conditionne la connaissance des composants d'état pour les agents vers lesquels on exporte ces composants. Dans ce cas, si la condition est vérifiée alors l'agent ne fait pas connaître les composants d'état.
3. XK exprime la combinaison des deux expressions K(composant.agent | condition) et I(composant.agent | ¬ condition).

Exemple :

```
CO-OPERATION CONSTRAINTS

STATE INFORMATION

K( Panne_a_venir.Reparateur | Panne_a_venir )
I( Etat.Controle | Panne_a_venir )
```

Dans cet exemple,

- L'agent fait connaître le composant d'état *Panne_a_venir* si celui-ci est à vrai.
- La seconde assertion exprime le fait que si le composant d'état *Panne_a_venir* est à vrai, l'agent ne permet pas à l'agent CONTROLE de percevoir le composant d'état *Etat*.

1.4 ETUDE DE CAS.

1.4.1 ORIGINE DE L'ENONCE.

Cet énoncé a été inspiré d'un énoncé développé dans le cadre de cours sur OBLOG par Joao Gouveia à l'I.S.T. (Institut Supérieur Technique) de Lisbonne. Il a été enrichi par une documentation sur le comportement des ascenseurs qui se trouve en annexe.

1.4.2 ENONCE

Le système d'ascenseur est composé d'une cabine d'ascenseur, d'un centre de contrôle, de plusieurs étages et d'un moteur. Ce système est actionné par plusieurs utilisateurs. De plus, chaque composant du système est équipé d'un détecteur de pannes pouvant survenir.

La cabine d'ascenseur comprend une porte, un boîtier de commandes et un senseur qui lui permet de connaître sa position. La porte de la cabine peut s'ouvrir ou se fermer et lorsqu'elle s'ouvre, elle reste ouverte pendant 5 secondes. Il est possible à l'utilisateur de prolonger l'ouverture de la porte de 5 autres secondes en actionnant le bouton d'ouverture de la porte sur le boîtier de commandes. La porte de l'ascenseur ne s'ouvre que si la cabine est arrêtée à une étage. Le boîtier de commande permet les actions suivantes : sélectionner une nouvelle destination, prolonger la durée d'ouverture de la porte et visualiser le numéro de l'étage où se trouve la cabine d'ascenseur.

Le centre de contrôle sert à synchroniser l'ouverture de la porte de la cabine et des portes des étages, à diffuser aux différents étages et à la cabine l'information concernant la position de l'ascenseur et à collecter les informations concernant les pannes à venir auprès des détecteurs de pannes. Lorsqu'une panne va se produire, le centre de contrôle le fait savoir au réparateur qui peut ainsi intervenir avant l'arrivée de la panne.

Chaque étage est équipé d'un boîtier de commandes. Le boîtier de commandes permet les actions suivantes : appeler la cabine d'ascenseur pour monter et appeler la cabine d'ascenseur pour descendre. Les boîtiers de commandes des premier et dernier étages sont différents car respectivement, ils ne permettent pas l'appel pour la descente et l'appel pour la montée. Il y a également une porte à chaque étage. Ces portes sont toutes fermées à l'exception de celle de l'étage où se trouve la cabine qui, elle, se comporte comme la porte de la cabine.

Le moteur de l'ascenseur peut soit faire monter le cabine, soit la faire descendre, soit maintenir la cabine en position.

La politique de choix de l'arrêt de l'ascenseur est basée sur la méthode de l'essuie-glace. Dans une première phase, l'ascenseur s'arrête à tous les étages supérieurs ayant été sélectionnés par les utilisateurs par un appel pour une montée ainsi qu'aux étages d'arrêt (sortie d'un utilisateur de la cabine). Dans cette phase, le premier étage choisi sera celui qui est le plus bas parmi les étages sélectionnés. Pour la deuxième phase, l'ascenseur s'arrête à tous les étages inférieurs ayant été sélectionnés par les utilisateurs par un appel pour une descente ainsi qu'aux étages d'arrêt. Dans cette phase, le premier étage choisi sera celui qui est le plus haut parmi les étages sélectionnés.

Lorsque la cabine d'ascenseur s'arrête à un étage, les portes de la cabines doivent impérativement s'ouvrir.

1.4.3 METHODOLOGIE APPLIQUEE A L'ETUDE DE CAS.

Dans cette section, nous vous présentons la méthodologie que nous avons appliquée pour réaliser la spécification.

1. Identifier les agents (acteurs qui ont une responsabilité).
2. Regrouper les concepts qui ne sont pas des agents autour d'un agent auquel ils se rattachent.
3. Définir à partir du regroupement les composants d'état.
4. Choisir un type de données correspondant le mieux à chaque composant et si besoin définir ce type.
5. Identifier les responsabilités des agents.
6. Retirer les actions de l'identification des responsabilités.
7. Définir les arguments des actions.
8. Décrire les effets des actions sur les composants d'état et les durées des actions.
9. Déterminer les interactions entre les agents.
10. Rendre visible les actions associées à ces interactions (perception et information).
11. Déterminer les informations qui doivent être partagées.
12. Rendre visible les composants d'état partagés (perception et information).
13. Définir la valeur initiale des composants d'état s'ils en ont une.
14. Définir des scénarii.
15. Généraliser les scénarii pour obtenir le comportement.
16. Choisir le style pour contrôler l'occurrence des actions.

17. Décrire le contrôle.

1.4.4 ETUDE DE L'UNIVERS DU DISCOURS

1.4.4.1 Agents

Quels sont les agents faisant partie de l'univers du discours ?

	Définition	Appellation
1	Centre de contrôle	CONTROLE
2	Moteur	MOTEUR
3	Utilisateurs	UTILISATEURS
4	Cabine	CABINE
5	Réparateur	REPARATEUR
6	Etages	ETAGES

1.4.4.2 Interactions entre agents

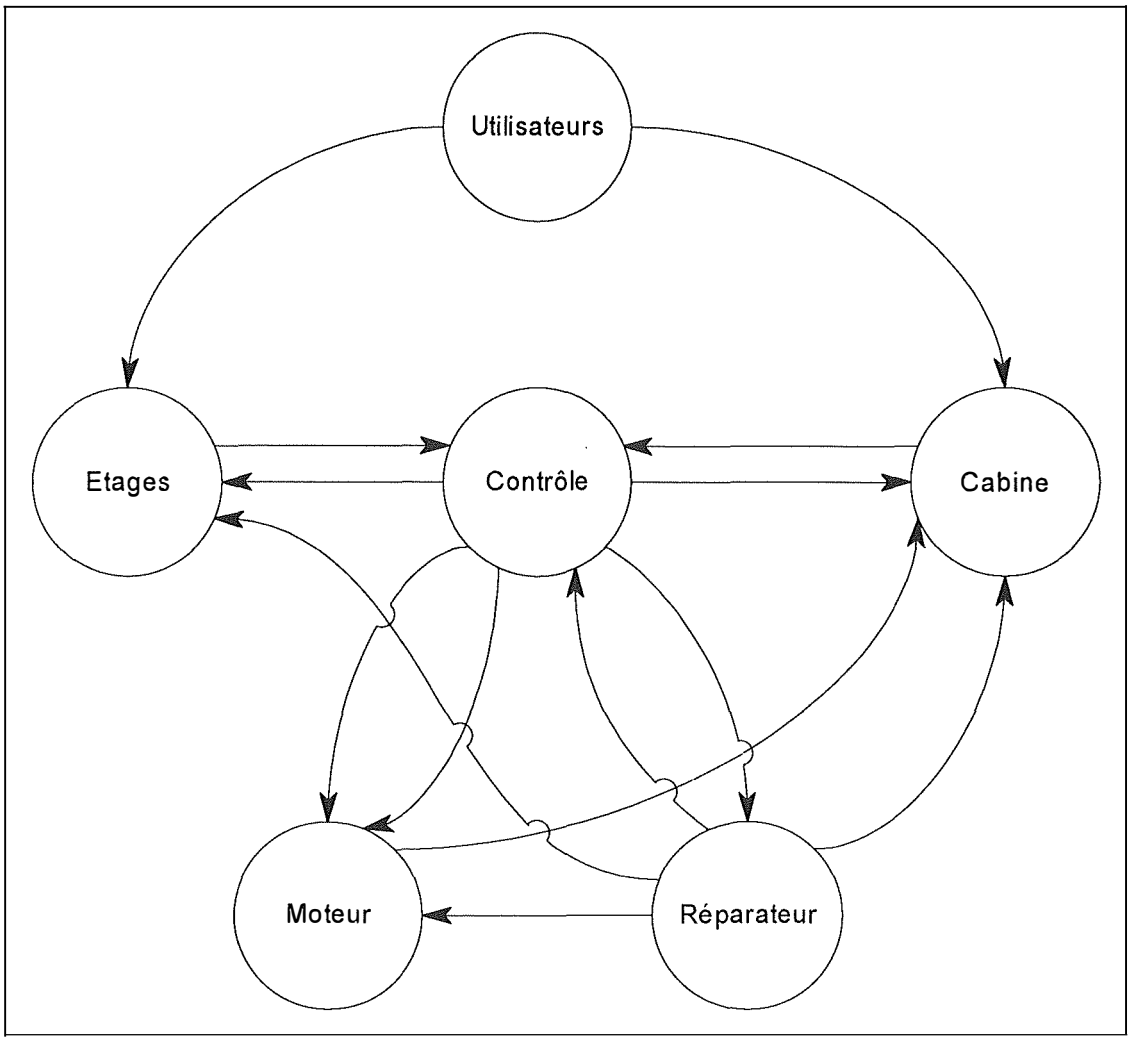


Figure 1-8 : Interactions entre agents.

1.4.4.3 Actions élémentaires

Quelles sont les actions élémentaires des différents agents ?

1.4.4.3.1 CONTROLE

- Commande d'ouverture de la porte de la cabine et de la porte de l'étage où est arrêtée la cabine.
- Commande de fermeture de la porte de la cabine et de la porte de l'étage où est arrêtée la cabine.
- Avertissement du réparateur d'une panne à venir.
- Mise en marche du moteur pour monter.
- Mise en marche du moteur pour descendre.
- Arrêt du moteur.
- Détection d'une panne potentielle.

1.4.4.3.2 MOTEUR

- Montée de la cabine.
- Descente de la cabine.
- Arrêt.
- Détection d'une panne potentielle.

1.4.4.3.3 UTILISATEURS

- Appel de la cabine pour monter.
- Appel de la cabine pour descendre.
- Ajout d'une destination depuis la cabine.
- Commande d'ouverture des portes depuis la cabine.

1.4.4.3.4 CABINE

- Détection d'une panne potentielle.

1.4.4.3.5 REPARATEUR

- Réparation préventive du centre de contrôle.
- Réparation préventive d'un étage.
- Réparation préventive de la cabine.

- Réparation préventive du moteur.

1.4.4.3.6 ETAGES

- Détection d'une panne potentielle.

1.4.4.4 Etats

1.4.4.4.1 CONTROLE

- Phase (ascendante, descendante).
- Indicateur de panne à venir.

1.4.4.4.2 MOTEUR

- Etat (montant, descendant, arrêté).
- Indicateur de panne à venir

1.4.4.4.3 UTILISATEURS

- Aucun.

1.4.4.4.4 CABINE

- Etat de la porte (ouverte ou fermée).
- Position actuelle de la cabine (étage).
- Liste des étages destinations des utilisateurs.
- Indicateur de panne à venir.

1.4.4.4.5 REPARATEUR

- Aucun.

1.4.4.4.6 ETAGES

- Numéro de l'étage.
- Etat de la porte (ouverte ou fermée).
- Présence d'un appel pour monter (sauf pour le dernier étage).
- Présence d'un appel pour descendre (sauf pour le premier étage).
- Indicateur de panne à venir.

1.4.5 SPECIFICATION.

Dans cette section, nous ne présentons que la société et les agents de la spécification dans leur version graphique. La spécification complète est présentée en annexe.

1.4.5.1 Société.

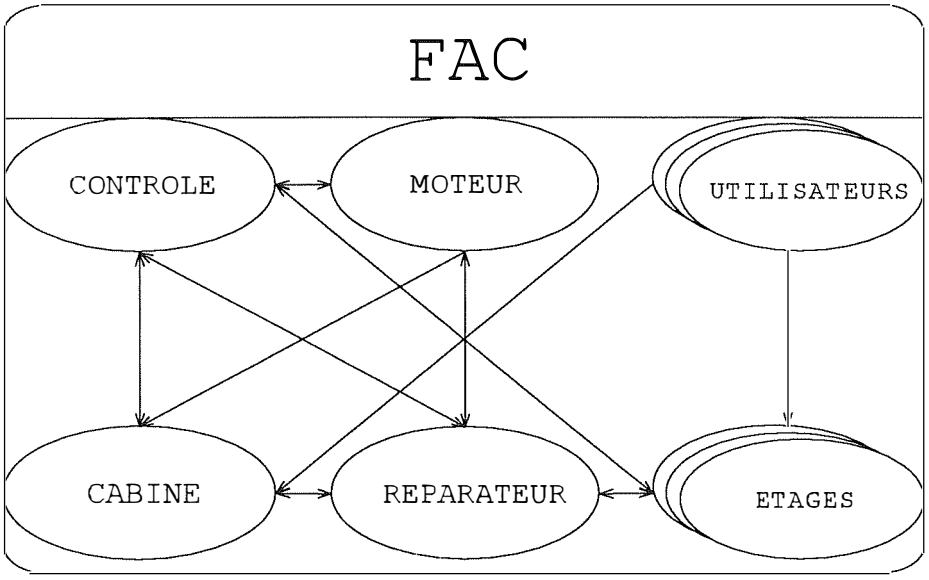


Figure 1-9 : Société

1.4.5.2 Agent CONTROLE.

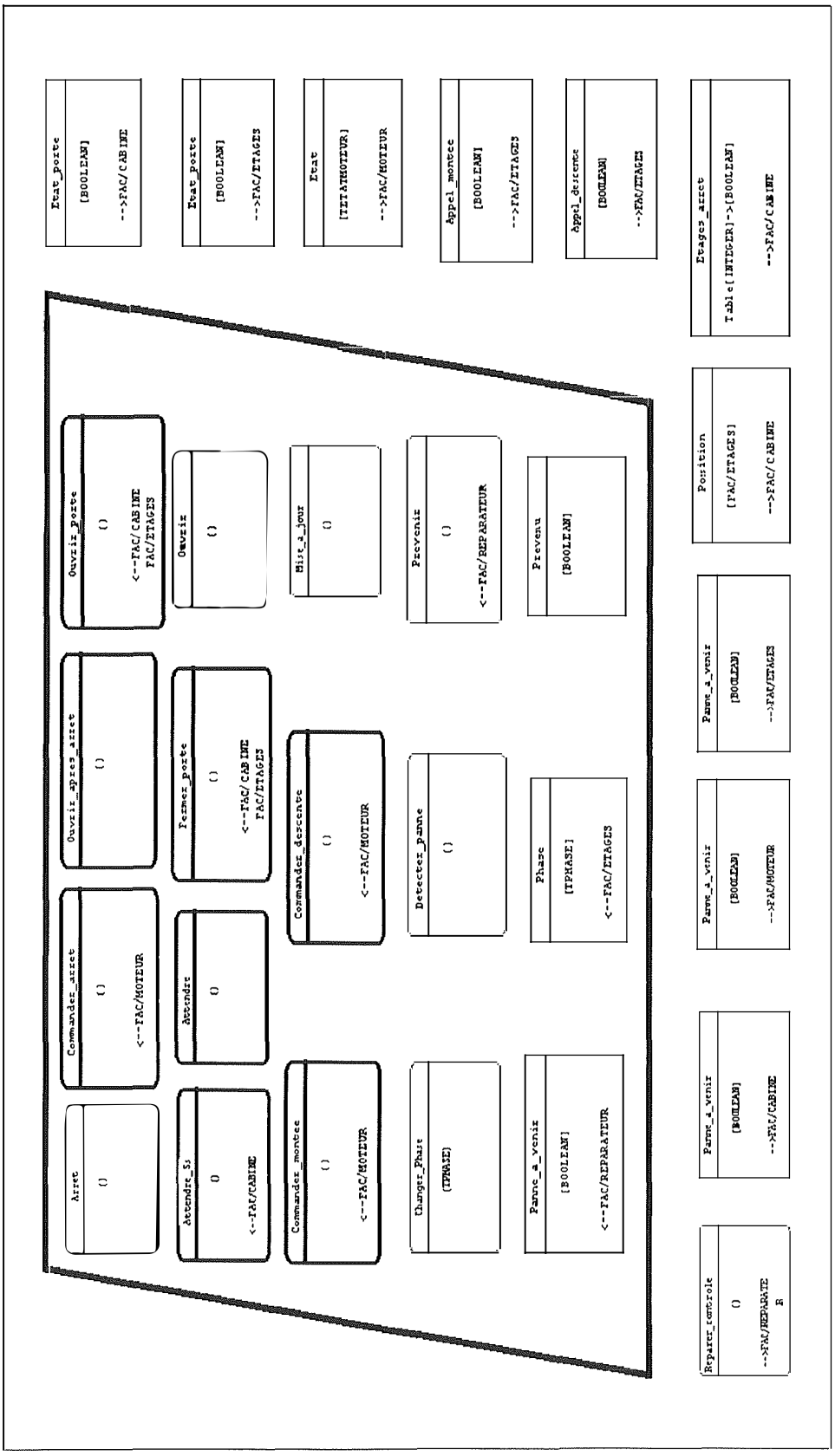


Figure 1-10 : Agent contrôle

1.4.5.2.1 Description des composants d'état.

`Phase` **`instance of`** `TPHASE`

Artefact introduit dans la spécification pour gérer les mouvements de la cabine.

`Panne_a_venir` **`instance of`** `BOOLEAN`

Composant d'état indiquant la présence ou non d'une panne à venir.

`Prevenu` **`instance of`** `BOOLEAN`

Artefact introduit dans la spécification pour limiter les occurrences inutiles de *prevenir*.

1.4.5.2.2 Description des actions.

`Arret`

Artefact introduit pour le contrôle des occurrences des actions *Commander_arret* et *Ouvrir_après_arret*.

`*Attendre`

Artefact introduit pour le contrôle des occurrences de l'action *Attendre_5s*.

`*Attendre_5s`

Action de temporisation pour permettre la sortie des utilisateurs de la cabine.

`Changer_Phase(TPHASE)`

Artefact introduit dans la spécification pour gérer les mouvements de la cabine.

`*Commander_arret`

Action commandant l'arrêt du moteur.

`Commander_descente`

Action commandant la mise en route du moteur pour descendre la cabine.

`Commander_montee`

Action commandant la mise en route du moteur pour monter la cabine.

`Detecter_panne`

Action de détection d'une panne potentielle.

`*Fermer_porte`

Action commandant l'ouverture des portes de la cabine et de l'étage où se trouve la cabine.

`Ouvrir`

Artefact introduit pour le contrôle des occurrences des actions *Ouvrir_porte*, *Attendre_5s*, *Attendre* et *Fermer_porte*.

`*Ouvrir_apres_arret`

Artefact introduit pour le contrôle des occurrences des actions *Ouvrir_porte*, *Attendre_5s*, *Attendre* et *Fermer_porte*.

`*Ouvrir_porte`

Action commandant la fermeture des portes de la cabine et de l'étage où se trouve la cabine.

`Prevenir`

Action prévenant le réparateur de la présence de pannes potentielles.

`Mise_a_jour`

Artefact introduit dans la spécification pour limiter les occurrences inutiles de *prevenir*.

1.4.5.3 Agent MOTEUR.

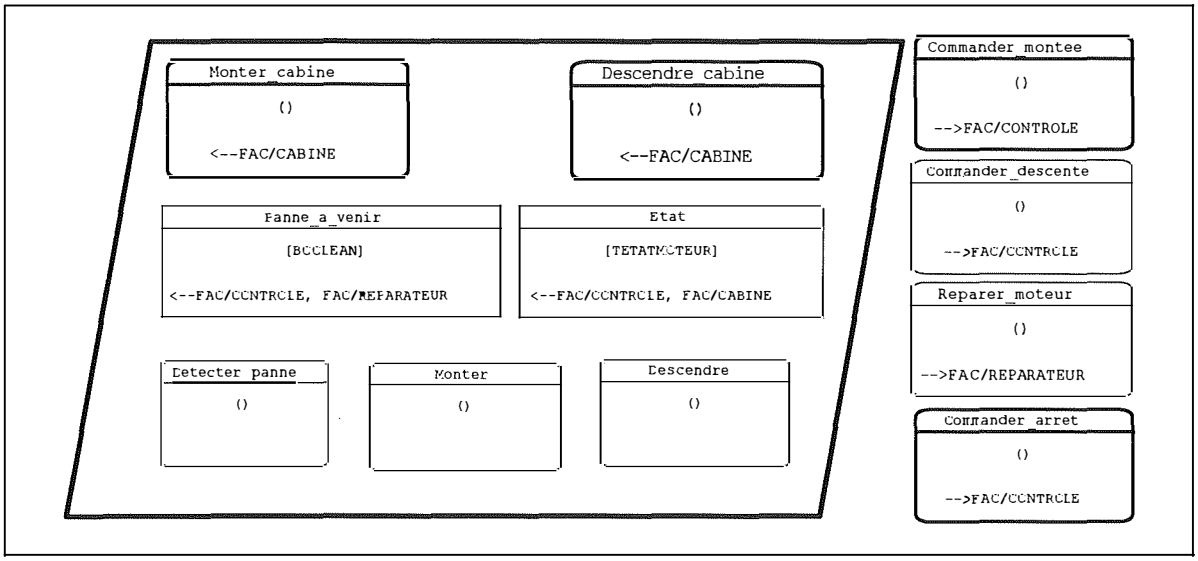


Figure 1-11 : Agent Moteur

1.4.5.3.1 Description des composants d'état.

`Etat` **`instance of`** `TETATMOTEUR`

Composant d'état contenant l'état du moteur (montant, descendant, arrêté).

`Panne_a_venir` **`instance of`** `BOOLEAN`

Composant d'état indiquant la présence ou non d'une panne à venir.

1.4.5.3.2 Description des actions.

`*Monter_cabine`

Action de montée de la cabine.

`*Descendre_cabine`

Action de descente de la cabine.

`Detecter_panne`

Action de détection d'une panne potentielle.

`Monter`

Artefact introduit pour le contrôle des occurrences des actions *Commander_montee* de contrôle et *Monter_cabine*.

`Descendre`

Artefact introduit pour le contrôle des occurrences des actions *Commander_descente* de contrôle et *Descendre_cabine*.

1.4.5.4 Agent UTILISATEURS.

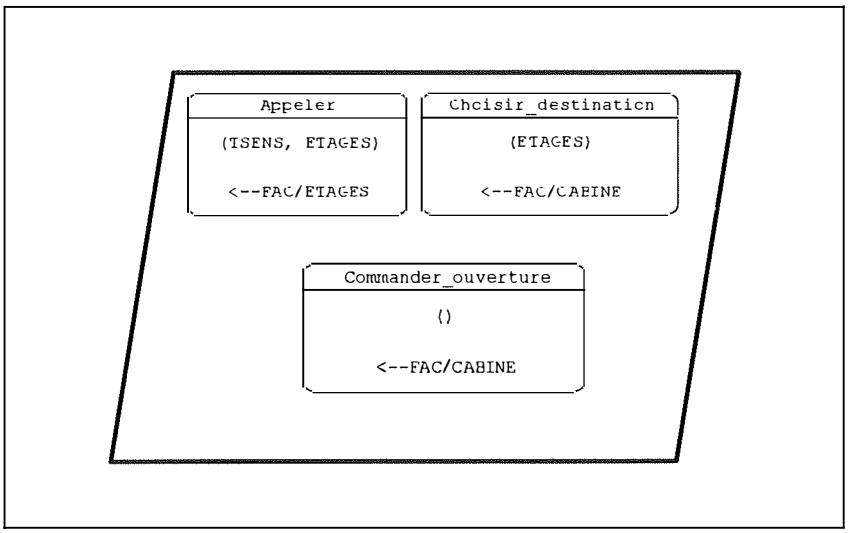


Figure 1-12 : Agent Utilisateurs

1.4.5.4.1 Description des composants d'état.

Pas de composants d'état.

1.4.5.4.2 Description des actions.

`Appeler(TSENS, ETAGES)`

Action d'appel de la cabine d'ascenseur pour monter ou descendre depuis un étage.

`Choisir_destination(ETAGES)`

Action d'enregistrement de l'étage de destination d'un utilisateur.

`Commander_ouverture`

Action de commande de prolongation de l'ouverture des portes de la cabine et de l'étage où se trouve la cabine.

1.4.5.5 Agent CABINE

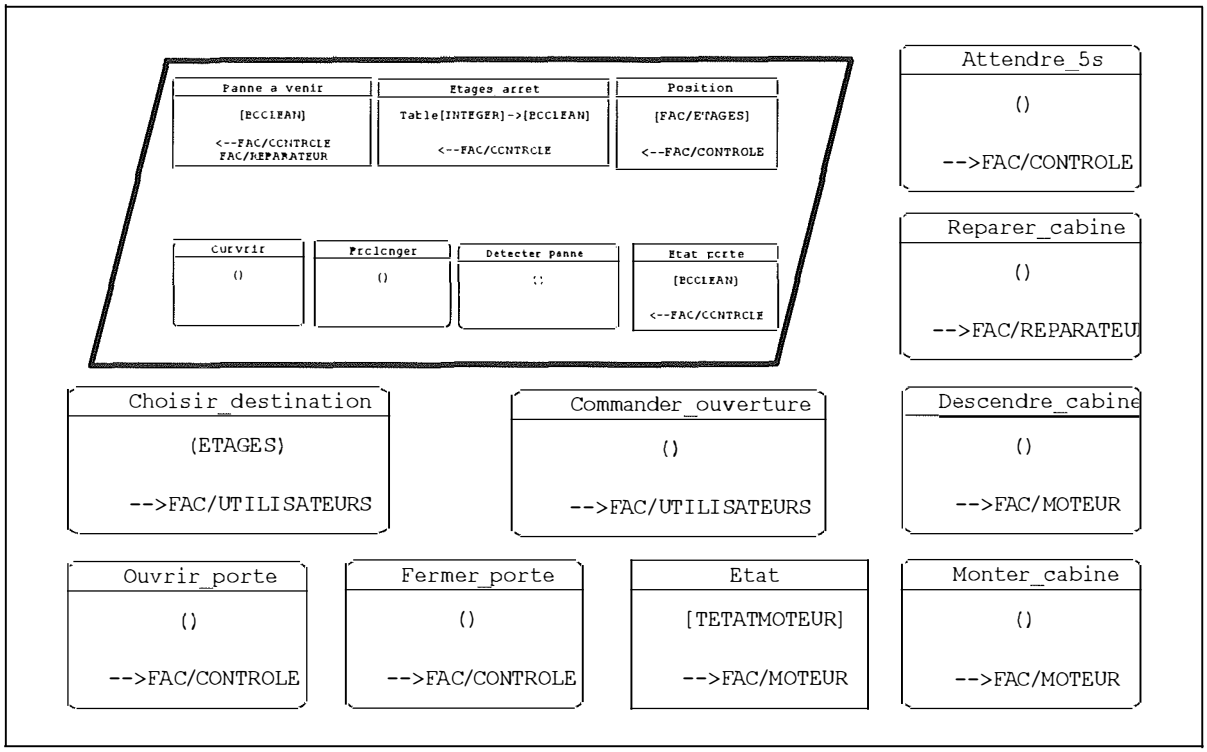


Figure 1-13 : Agent cabine

1.4.5.5.1 Description des composants d'état.

`Etat_porte` **`instance of`** `BOOLEAN`

Composant d'état servant pour l'état de la porte (Vrai ⇒ porte ouverte)

`Position` **`instance of`** `ETAGES`

Composant d'état indiquant la position actuelle de la cabine.

`Panne_a_venir` **`instance of`** `BOOLEAN`

Composant d'état indiquant la présence ou non d'une panne à venir.

`Etages_arret` **`table of`** `BOOLEAN` **`indexed by`** `ETAGES`

Composant d'état servant à indiquer à quels étages la cabine doit s'arrêter pour laisser sortir un utilisateur.

1.4.5.5.2 Description des actions.

`Detecter_panne`

Action de détection d'une panne potentielle.

`Ouvrir`

Artefact introduit pour le contrôle des occurrences des actions *Commander_ouverture* de utilisateurs et *Ouvrir_porte* de contrôle.

Prolonger

Artefact introduit pour le contrôle des occurrences des actions *Commander_ouverture* de utilisateurs et *Attendre_5s* de contrôle.

1.4.5.6 Agent REPARATEUR

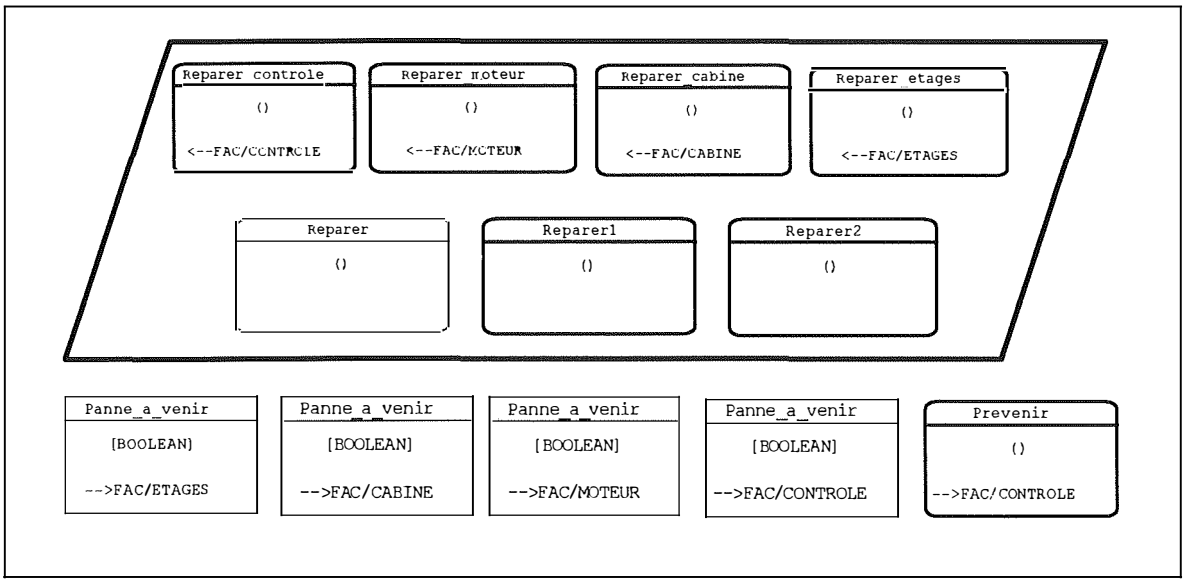


Figure 1-14 : Agent Réparateur

1.4.5.6.1 Description des composants d'état.

Pas de composants d'état.

1.4.5.6.2 Description des actions.

*Reparer_contrôle

Action de réparation de l'élément contrôle.

*Reparer_moteur

Action de réparation de l'élément moteur.

*Reparer_cabine

Action de réparation de l'élément cabine.

*Reparer_etages

Action de réparation des éléments étages.

*Reparer1

Artefact introduit pour le contrôle des occurrences de l'action *Reparer2*.

*Reparer2

Artefact introduit pour le contrôle des occurrences des actions *Reparer_controle, Reparer_moteur, Reparer_cabine* et *Reparer_etages*.

Reparer

Artefact introduit pour le contrôle des occurrences des actions *Prevenir* de contrôle et *Reparer1*.

1.4.5.7 Agent ETAGES

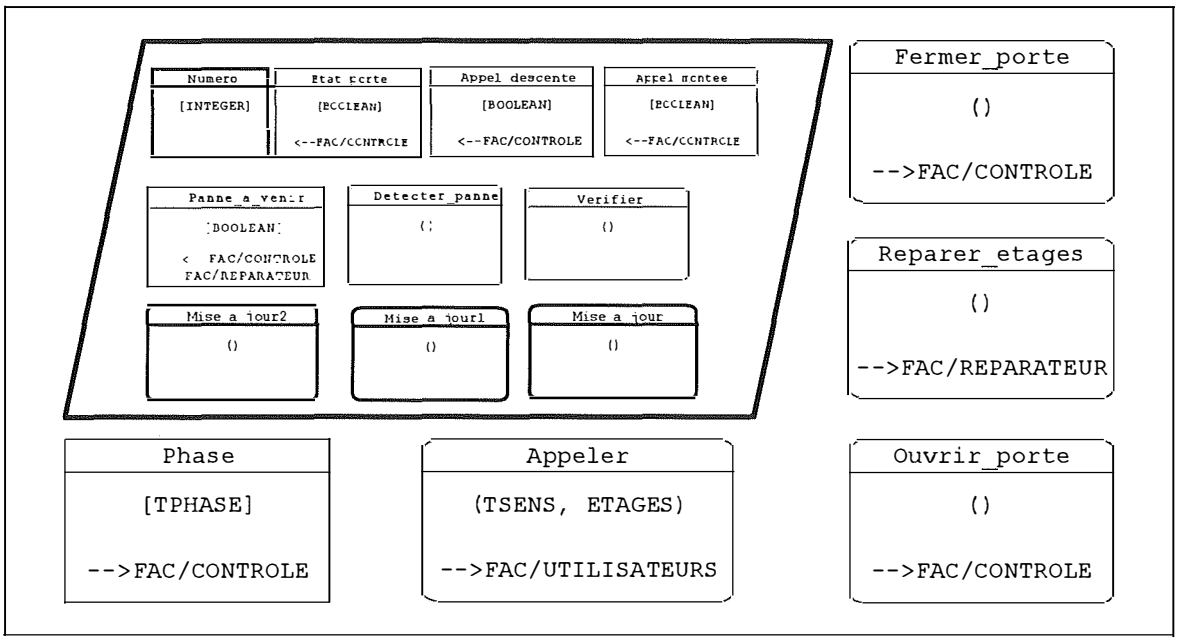


Figure 1-15 : Agent Etages

1.4.5.7.1 Description des composants d'état.

*Numero **instance of** INTEGER

Composant d'état constant indiquant le numéro de l'étage.

Etat_porte **instance of** BOOLEAN

Composant d'état servant pour l'état de la porte (Vrai ⇒ porte ouverte)

Appel_montee **instance of** BOOLEAN

Composant d'état indiquant la présence d'un appel de montée.

Appel_descente **instance of** BOOLEAN

Composant d'état indiquant la présence d'un appel de descente.

Panne_a_venir **instance of** BOOLEAN

Composant d'état indiquant la présence ou non d'une panne à venir.

1.4.5.7.2 Description des actions.

`Detecter_panne`

Action de détection d'une panne potentielle.

`Verifier`

Artefact introduit pour le contrôle des occurrences des actions *Ouvrir_porte* de contrôle et *Mise_a_jour*.

`*Mise_a_jour1`

Artefact introduit pour la mise à jour conditionnelle du composant d'état *Appel_montee*.

`*Mise_a_jour2`

Artefact introduit pour la mise à jour conditionnelle du composant d'état *Appel_descente*.

`*Mise_a_jour`

Artefact introduit pour le contrôle des occurrences des actions *Mise_a_jour1* et *Mise_a_jour2*.

1.5 COMMENTAIRES SUR LA SPECIFICATION.

Dans cette section, nous présentons la description des opérations, des composants d'état et des actions ainsi que différents commentaires sur l'architecture des agents et les styles utilisés.

1.5.1 DESCRIPTION DES OPERATIONS.

Les opérations qui ont été développées dans le cadre de cette spécification ont pour but de mettre en séquence les différentes instances de l'agent ETAGES. Ceci est réalisé au niveau des opérations parce que leur portée est globale.

Dans un premier temps, nous avons décrit des constantes pour définir le premier et le dernier élément de la séquence d'ETAGES. La seule contrainte à ce niveau était que ces constantes soient différentes.

Ensuite, nous avons développé une opération pour mettre en séquence les instances et nous nous sommes rendu compte qu'il était nécessaire de définir des opérations renvoyant le précedent et le suivant d'un élément.

1.5.1.1 Opération Avant.

```
Avant : ETAGES × ETAGES → BOOLEAN
Avant( a , b ) = val
with [val = FALSE ⇐ b = PremierEtage]
   ∧ [val = TRUE ⇔ a = Prec(b) ∨ Avant(a,Prec(b))]
```

Cette opération renvoie :

- la valeur FALSE si le second argument de l'opération est le premier élément de la séquence.
- la valeur TRUE si et seulement si le premier argument est le précédent du second arguent dans la séquence ou s'il est avant le précédent de ce même second argument.

1.5.1.2 Opération Prec.

```
Prec : ETAGES → ETAGES*
Prec (a) = val
with (val = UNDEF ⇔ a = PremierEtage)
   ∧ (¬∃ b : a <> b ∧ Prec(a)=Prec(b))
```

Cette opération retourne :

- la valeur UNDEF si l'argument de l'opération est le premier élément de la séquence.
- un étage tel qu'il n'existe pas d'étage distinct de l'argument ayant le même précédent.

1.5.1.3 Opération Suiv.

```
Suiv : ETAGES → ETAGES*
Suiv ( a ) = val
with (val = UNDEF ⇔ a = DernierEtage)
 ∧ (¬∃ b : a <> b ∧ Suiv(a)=Suiv(b))
 ∧ (x <> PremierEtage ⇒ x = Suiv(Prec(x)))
```

Cette opération retourne :

- la valeur UNDEF si l'argument de l'opération est le dernier élément de la séquence.
- un étage tel qu'il n'existe pas d'étage distinct de l'argument ayant le même suivant et tel que tous les étages différents du premier étage sont le suivant de leur précédent.

1.5.2 EXPLICATION DE L'ARCHITECTURE DES AGENTS.

Lors de la réalisation de cette spécification, nous avons mis en évidence deux possibilités pour ce qui est de l'architecture des agents. Dans une première architecture, développée dans ce mémoire, il y a 6 agents qui sont CONTROLE, MOTEUR, CABINE, ETAGES, UTILISATEURS et REPARATEURS et leur choix en temps qu'agent se justifie par le fait qu'ils ont chacun des responsabilités au sein du système.

Dans une seconde architecture, assez similaire, seul l'agent ETAGES était modélisé différemment. Au lieu d'être un agent avec plusieurs instances possibles, il était représenté par 6 agents différents à instance unique (ETAGE_1, ETAGE0, ..., ETAGE4). La raison de ce choix était de pouvoir définir des contraintes sur les numéros de ces étages (unicité du numéro afin d'avoir une séquence), ce qui n'est pas possible dans la première architecture puisqu'ALBERT II ne permet pas d'exprimer des contraintes globales sur les agents. Ensuite, nous avions construit un type ETAGES qui était l'union (ou la collection) de ces différents agents. Et nous voulions l'utiliser comme l'agent défini dans la première version. Malheureusement, le langage ALBERT II ne supporte pas le « polymorphisme » or cela était nécessaire pour continuer sur cette voie. Nous avons donc dû abandonner cette version et reprendre la version précédente avec ses désavantages.

Il a donc fallu trouver un moyen pour mettre en séquence les instances d'étage. Ceci a été réalisé grâce aux opérations présentées à la section précédente.

1.5.3 STYLES POSSIBLES POUR DECRIRE LE COMPORTEMENT D'UN AGENT.

Le comportement d'un agent se décrit en terme d'actions qui peuvent survenir et le contrôle des occurrences est géré de trois façons en ALBERT II : « *State Behaviour* », « *Etat-Transition* » et « *Action Composition* ». Nous présentons dans l'exemple qui suit les trois styles de modélisations appliqués à un même problème. Dans cet exemple, nous ne modélisons que les éléments pertinents.

1.5.3.1 Spécification orientée « *State Behaviour* ».

Enoncé :

Des utilisateurs peuvent ouvrir une porte. Si cette porte est ouverte à un moment donné, alors à un moment dans le futur, elle devra être fermée. La seule façon d'avoir la porte fermée est de réaliser l'action fermer.

```
AGENT CONTROLE

STATE COMPONENTS
Ouvert instance of BOOLEAN

ACTIONS
Fermer

BASIC CONSTRAINTS
INITIAL VALUATION
Ouvert = False

LOCAL CONSTRAINTS
STATE BEHAVIOUR
Ouvert ⇒ ◊¬Ouvert

EFFECTS OF ACTION
U.Ouvrir : Ouvert := True
Fermer : Ouvert := False
```

Commentaires :

Il est à remarquer dans ce style que le contrôle de l'action *Fermer* est piloté par la contrainte d'évolution portant sur le composant d'état *Ouvert*. En effet, l'action *Fermer* se produira, si à un moment donné *Ouvert* est à vrai. La raison de cette occurrence est que cette action est la seule à mettre la valeur de ce composant d'état à faux.

1.5.3.2 Spécification orientée « *Etat-Transition* ».

Enoncé :

Des utilisateurs peuvent ouvrir une porte. Si cette porte a été ouverte à un moment donné, alors le contrôle doit la fermer.

```
AGENT CONTROLE

<Idem spécification précédente>

LOCAL CONSTRAINTS

EFFECTS OF ACTION

U.Ouvrir : Ouvert := True
Fermer : Ouvert := False

CAPABILITY

XO( Fermer | Ouvert )
```

Commentaires :

Dans ce style de contrôle, il faut introduire un artefact au niveau des composants d'état. En effet, la seule raison d'être du composant *Ouvert* est de permettre ce contrôle. Ce contrôle est réalisé grâce à la contrainte d'obligation exclusive qui force l'occurrence de l'action si la condition est vérifiée c'est-à-dire si l'agent se trouve dans un certain état (exprimé par la condition), il doit réaliser une certaine action.

1.5.3.3 Spécification orientée « *Action Composition* ».

Enoncé :

Des utilisateurs peuvent ouvrir une porte. Si cette action a lieu, elle doit toujours être suivie d'une action de fermeture.

AGENT CONTROLE

ACTIONS

```
Sequence
*Fermer
```

LOCAL CONSTRAINTS

ACTION COMPOSITION

```
Sequence ↔ U.Ouvrir ; Fermer
```

AGENT UTILISATEUR

ACTIONS

```
Ouvrir                    → CONTROLE*
```

Commentaires :

Dans ce cas-ci, l'artefact n'a pas été introduit au niveau des composants d'état mais au niveau des actions. En effet, la seule raison d'être de l'action *Sequence* est de permettre le contrôle.

1.5.4 STYLES UTILISES.

1.5.4.1 Agent CONTROLE.

Le style utilisé pour la politique de gestion des arrêts est orienté « *Etat-Transition* ». Ceci se justifie par le fait que les autres styles ne sont pas applicables :

- En utilisant le style « *Action Composition* », les scénarii générés sont premièrement trop complexes et deuxièmement ne permettent pas d'exprimer certaines contraintes car les réactions de cet agent sont conditionnées par un trop grand nombre de composants d'état externes.
- Le style « *State Behaviour* » ne convient pas puisqu'il ne contraint que des composants d'état propres à l'agent alors qu'ici ce sont des composants externes qui font réagir l'agent.

Le graphe d' « état-transition » du composant d'état *Phase* est présenté à la Figure 1-16. Les différentes transitions et leur condition de réalisation sont expliquées ci-dessous :

1. STABLE → UP : Il existe un étage supérieur à l'étage où se trouve la cabine pour lequel il y a soit un appel de montée soit un arrêt prévu.
2. STABLE → DOWN : Il existe un étage inférieur à l'étage où se trouve la cabine pour lequel il y a soit un appel de descente soit un arrêt prévu.
3. STABLE → PREPAREUP : Il existe un étage inférieur à l'étage où se trouve la cabine pour lequel il y a un appel de montée.
4. STABLE → PREPAREDOWN : Il existe un étage supérieur à l'étage où se trouve la cabine pour lequel il y a un appel de descente.
5. DOWN → PREPAREUP : Il existe un étage inférieur à l'étage où se trouve la cabine pour lequel il y a un appel de montée et il n'y a ni appel de descente ni arrêt prévu pour ces mêmes étages ainsi que pour l'étage où se trouve la cabine.
6. UP → PREPAREDOWN : Il existe un étage supérieur à l'étage où se trouve la cabine pour lequel il y a un appel de descente et il n'y a ni appel de montée ni arrêt prévu pour ces mêmes étages ainsi que pour l'étage où se trouve la cabine.
7. PREPAREDOWN → DOWN : Il n'existe pas d'étages supérieurs pour lesquels il y a un appel de descente.
8. PREPAREUP → UP : Il n'existe pas d'étages inférieurs pour lesquels il y a un appel de montée.
9. DOWN → STABLE : Il n'y a pas d'appel de montée, d'appel de descente ou d'arrêt prévu pour tous les étages ou bien il existe un étage supérieur pour lequel il y a un appel de descente et il n'existe aucun appel de montée ni d'arrêt prévu pour tous les étages et il n'y a pas d'appel de descente pour les étages inférieurs ainsi que pour l'étage où se trouve la cabine.
10. UP → STABLE : Il n'y a pas d'appel de montée, d'appel de descente ou d'arrêt prévu pour tous les étages ou bien il existe un étage inférieur pour lequel il y a un appel de montée et il n'existe aucun appel de descente ni d'arrêt prévu pour tous les étages et il n'y a pas d'appel de montée pour les étages supérieurs ainsi que pour l'étage où se trouve la cabine.
11. DOWN → UP : Il n'y a pas d'appel de montée, d'appel de descente ou d'arrêt prévu pour les étages inférieurs et il existe un étage supérieur pour lequel il y a un appel de montée et il n'existe pas d'appel de descente ou d'arrêt prévu pour l'étage où se trouve la cabine.

12. UP → DOWN : Il n'y a pas d'appel de montée, d'appel de descente ou d'arrêt prévu pour les étages supérieurs et il existe un étage inférieur pour lequel il y a un appel de descente et il n'existe pas d'appel de montée ou d'arrêt prévu pour l'étage où se trouve la cabine.

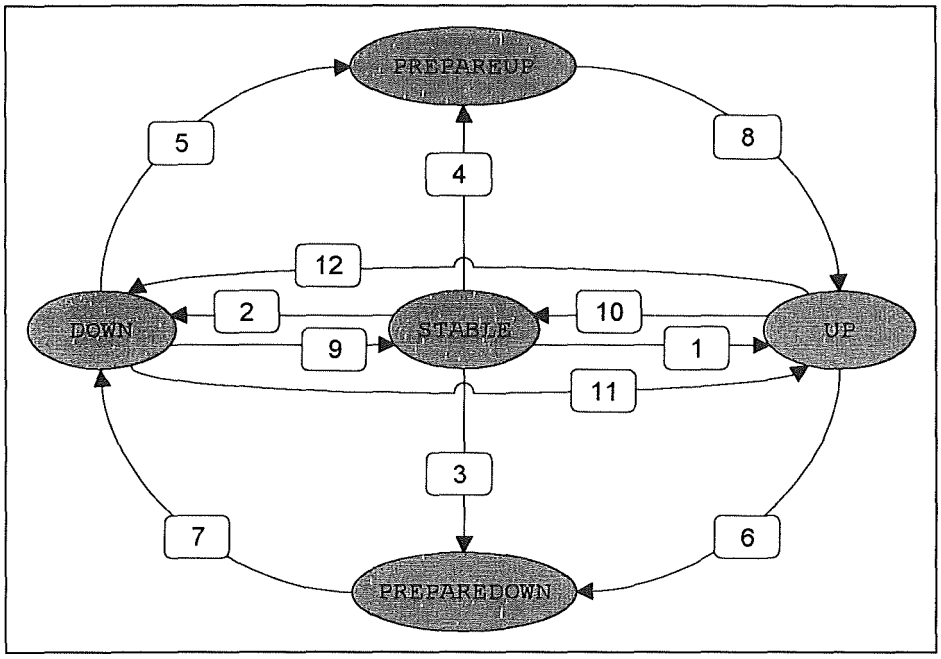


Figure 1-16 : Graphe des états et des transitions du composant d'état Phase.

Pour ce qui est des actions sur les portes, le style « *Action Composition* » a été préféré car les scénarii possibles sont assez peu nombreux et que la séquence ouverture, attente et fermeture est typiquement modélisable dans ce style.

1.5.4.2 Agent MOTEUR.

Cet agent est catégorisé comme réactif puisqu'il ne prend aucune « décision ». Les styles « *Action Composition* » et « *State Behaviour* » conviennent bien pour modéliser ce type d'agent. Dans ce cas, le style « *Action Composition* » a été préféré parce qu'il évite l'ajout d'un composant d'état artificiel comme décrit dans le paragraphe 1.5.3.

1.5.4.3 Agent UTILISATEURS.

Cet agent est externe au système et ne fait que déclencher des actions de façon indéterministe. Il n'est de ce fait pas nécessaire de spécifier des clauses particulières quant à son comportement.

1.5.4.4 Agent CABINE.

Nous avons également affaire à un agent réactif comme pour l'agent MOTEUR. Donc, les remarques faites au sujet de cet agent restent valables ici.

1.5.4.5 Agent REPARATEUR.

Cet agent est externe et réactif. Nous avons opté pour un style « *Action Composition* » dans le but encore une fois d'éviter d'introduire des composants d'état supplémentaires.

1.5.4.6 Agent ETAGES.

Ici aussi, comme pour les agents MOTEUR et CABINE, nous pouvons parler d'agent réactif mais à cette réaction s'ajoute une part de décision. C'est pourquoi nous rencontrons le style « *Action Composition* » comme pour les autres agents réactifs. La part de décision est gérée par les contraintes de « *Capability* ».

1.6 COMMENTAIRES SUR L'EDITEUR POUR WINDOWS 95.

Tout d'abord, il est à souligner que l'éditeur est d'une aide précieuse pour le développement de spécifications en ALBERT II. Il est convivial bien qu'il soit encore en phase de développement. Et la présence de certains composants (bouton Check, ...) montre que certaines fonctionnalités y seront ajoutées. Dans le but d'améliorer encore cet éditeur, nous nous permettons quelques remarques.

- Dans le premier écran avec l'arborescence d'une spécification (Figure 1-17), il serait intéressant d'avoir un outil permettant la réorganisation des sociétés, des agents, des états et des actions. Car, si pour des raisons de lisibilité, un utilisateur désire trier les sociétés, les agents, les composants d'état ou les actions par ordre alphabétique ou sur d'autres critères, ce travail est plus que laborieux pour l'instant puisque seuls les « couper-coller » permettent ces opérations.

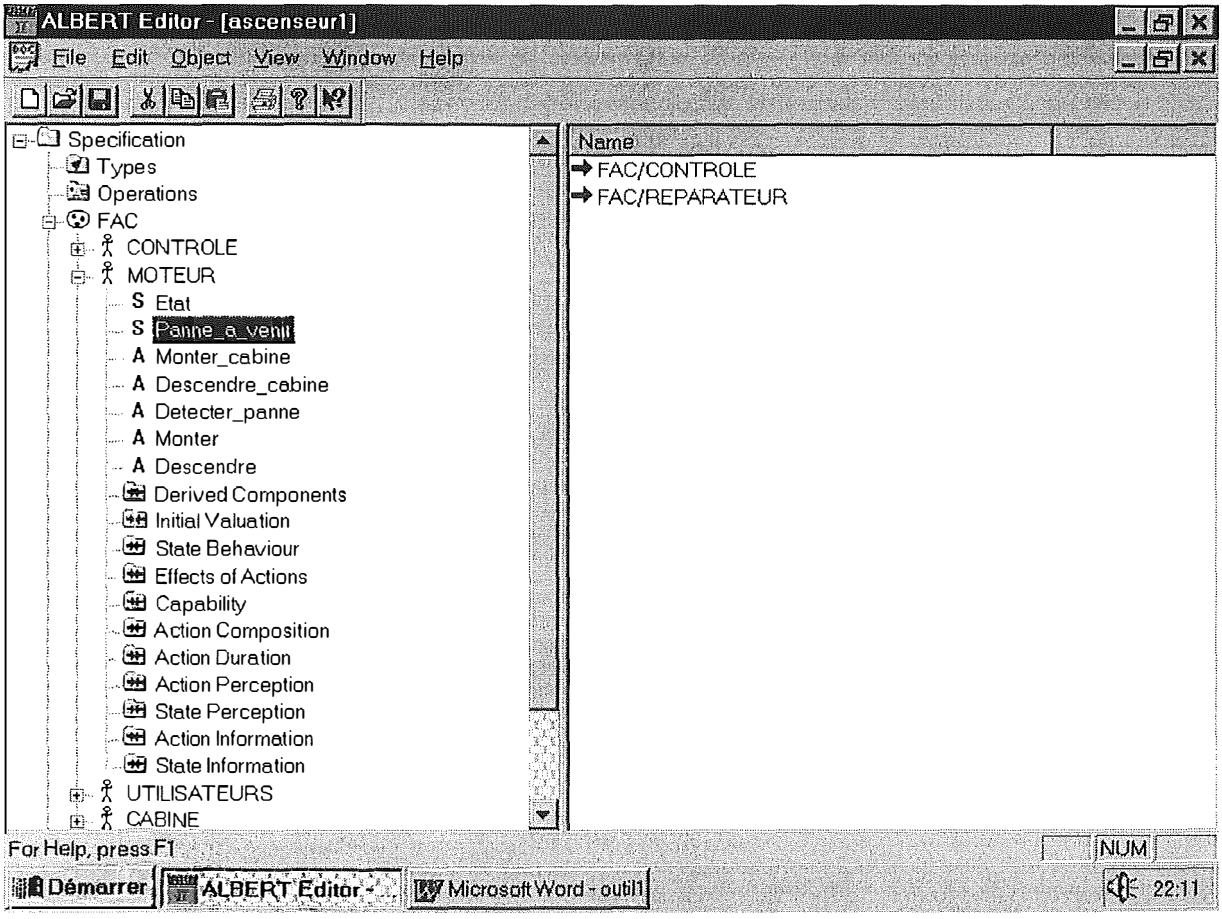


Figure 1-17 : Ecran d'arborescence d'une spécification ALBERT II.

- De plus, la structure de l'arborescence ne nous paraît pas adéquate. Pourquoi ne pas avoir gardé la structure présentée dans la syntaxe textuelle ? Nous reprenons cette structure dans la Figure 1-18.

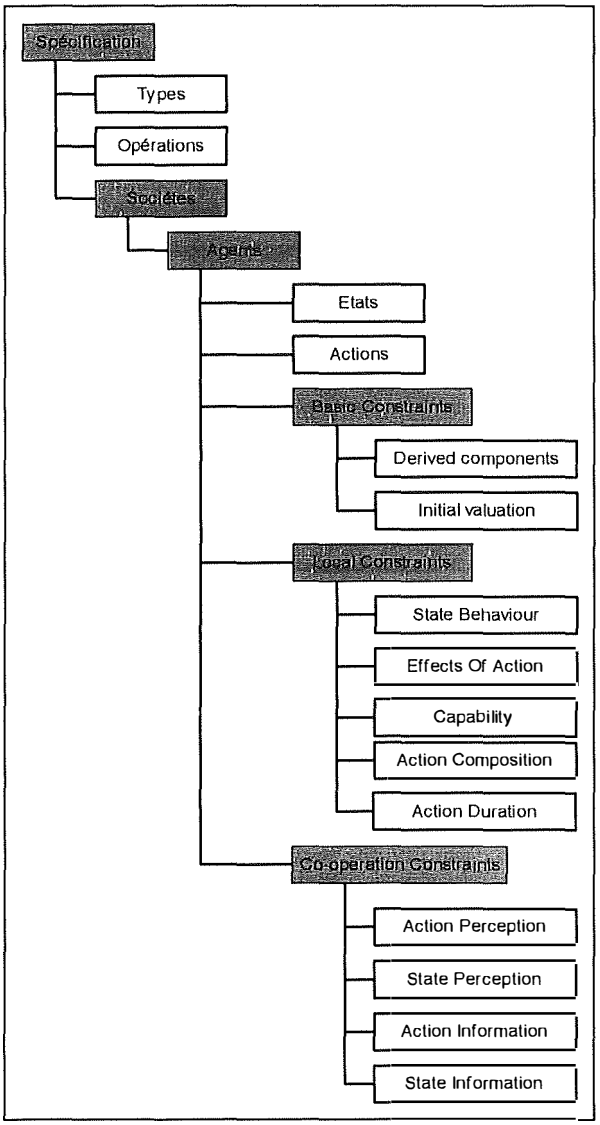


Figure 1-18 : Arborescence d'une spécification.

- Les différentes boîtes de dialogue pour les contraintes (Figure 1-19) ont des titres peu expressifs puisqu'elles ont toutes le même titre (*Constraint*). Ne serait-il pas intéressant de savoir de quel type de contrainte il s'agit ?
- De plus dans ces différentes boîtes de dialogue, il serait également utile d'ajouter des outils permettant l'aide à l'écriture des contraintes comme, par exemple, un « combo box » contenant les différents composants d'état pour les contraintes de tout type et un autre contenant les actions pour les contraintes de type *Effect Of Action, Action Composition, Action Du-*

ration, Action Perception et *Action Information*. Ceci éviterait à l'utilisateur de devoir connaître l'orthographe exacte des actions et des composants d'état.

- On pourrait également avoir une fenêtre avec tous les symboles utilisés dans les expressions ALBERT.

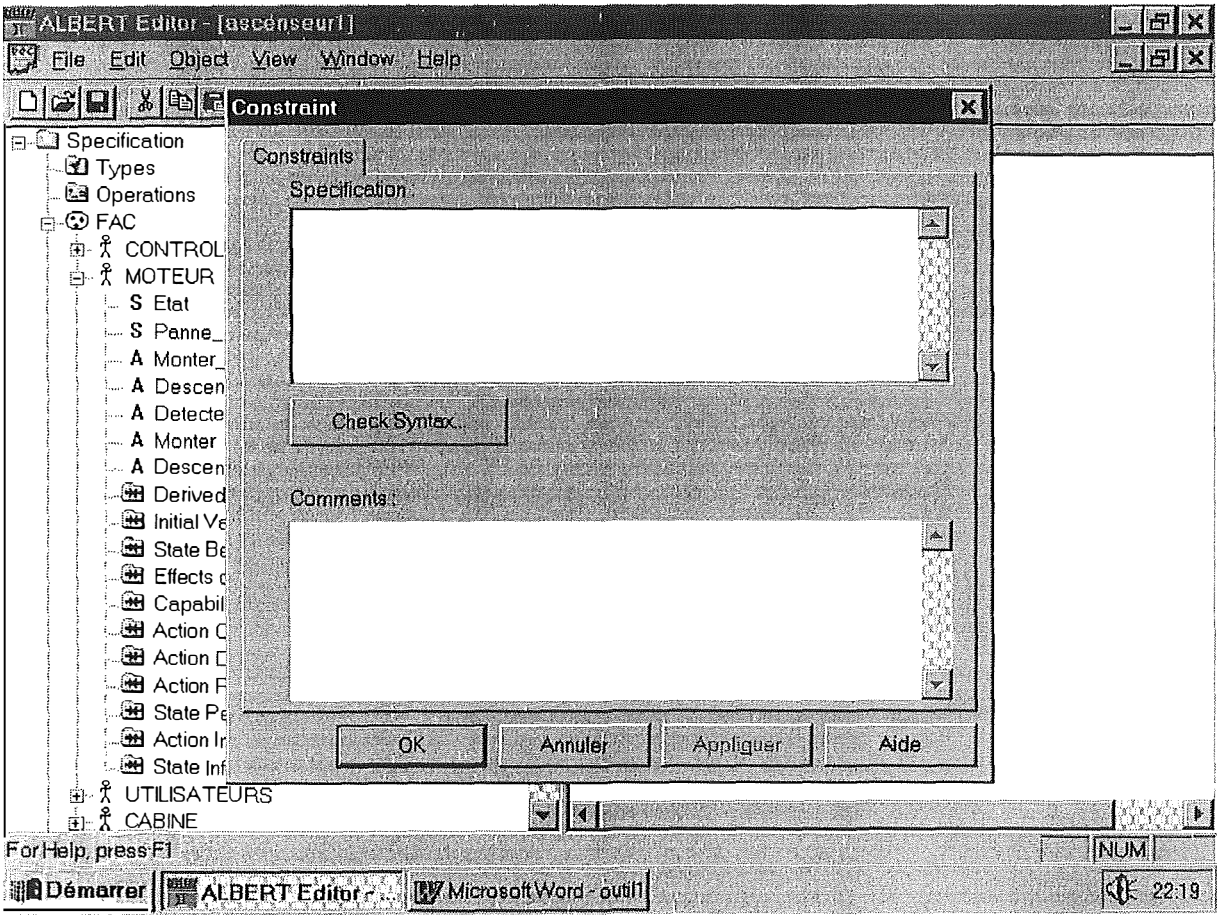


Figure 1-19 : Ecran avec une boîte de dialogue.

- Dans la fenêtre affichant la représentation graphique d'une spécification ALBERT, la figure représentant une action ou un composant d'état qui est importé depuis une classe d'agent (dont le nombre d'instances peut être supérieur à un) est la même que pour une classe à instance unique. Or ce qui est observé par l'agent importateur, c'est une série d'actions ou de composants d'état.
- Toujours dans cette fenêtre, il serait aussi intéressant d'avoir des outils pour aligner les figures, les dimensionner comme dans ABC Flowchart 4.0©.
- Pour ce qui est des copier-coller vers un document *word* ou autre, dans la version actuelle, il n'est possible que de copier la totalité des composants d'état et des actions. Or, il est également intéressant de copier une partie de ces actions et de ces composants.

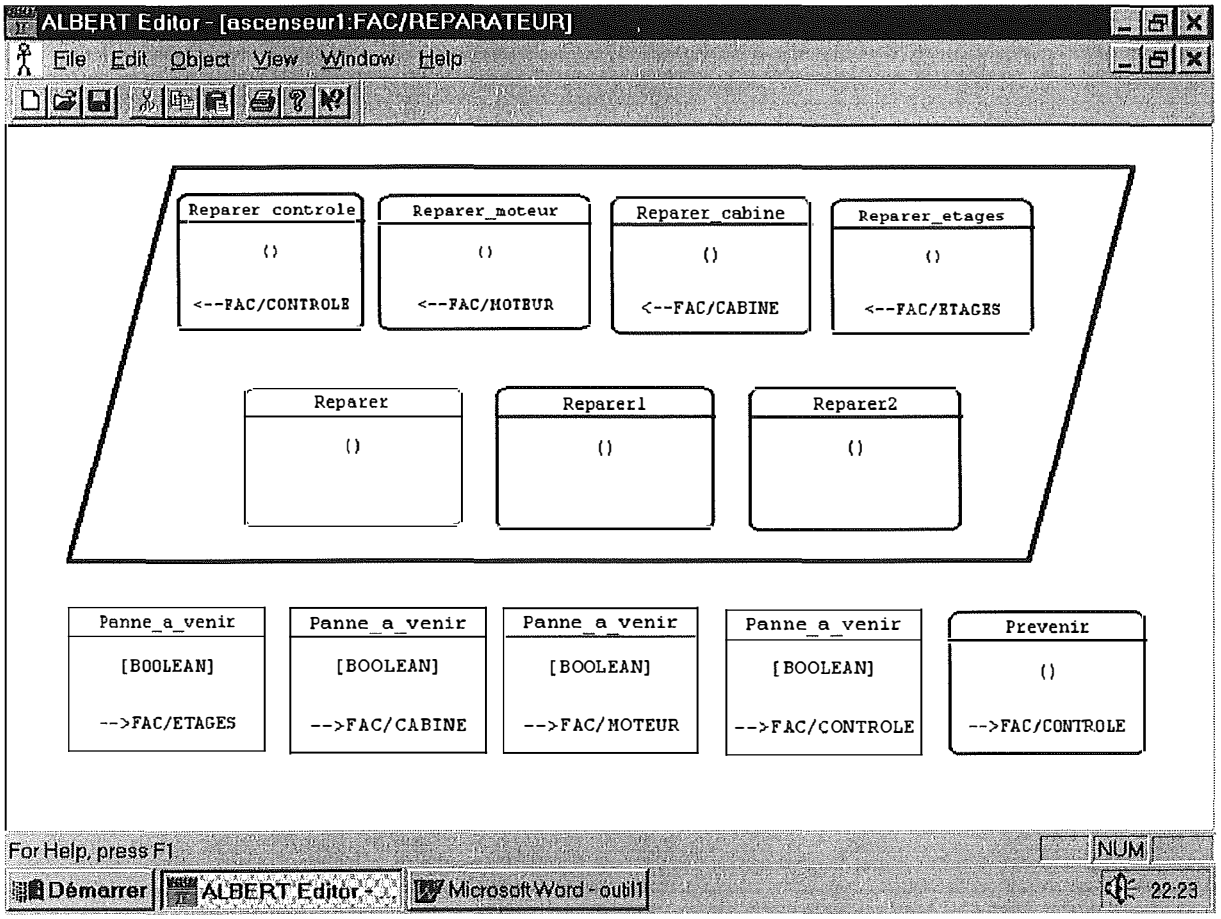


Figure 1-20 : Représentation graphique des composants et des actions d'un agent.

1.7 COMMENTAIRES SUR LE LANGAGE.

Lors de la réalisation de la spécification de l'ascenseur, plusieurs limites du langage sont apparues tel que l'absence de contraintes globales et de « polymorphisme ».

1.7.1 CONTRAINTES GLOBALES.

Les contraintes globales sont très souvent utilisées pour les classes d'agent à instances multiples. Elles permettent de définir entre autre des identifiants, des relations vérifiées par l'ensemble de la classe, ...

Or, les outils pour exprimer ce type de contraintes n'existent pas en ALBERT II. Ils sont présentés dans la littérature OO sous les concepts de *Class Operations* et *Class Attributes* [RUM91].

Par exemple, il est impossible d'exprimer:

- Tous les agents de la classe Etages ont un numéro différent.
- La classe d'agent Etages ne comporte que 6 instances au maximum.

1.7.2 PSEUDO-POLYMORPHISME.

Le concept de polymorphisme est souvent lié à celui d'héritage. Il se définit comme étant le mécanisme permettant d'associer plusieurs implémentations d'actions avec le même identifiant (nom de l'action).

Nous pourrions appliquer ce mécanisme non pas aux actions mais aux composants d'état pour pouvoir utiliser les collections comme nous utilisons les agents.

Prenons un exemple : Nous avons 3 agents et nous créons leur union. Ce qui nous aimerions faire, c'est de pouvoir traiter cet ensemble comme s'il s'agissait d'un agent.

Exemple :

```
CONSTRUTED TYPES
ETAGES = Union[ETAGE0,ETAGE1,ETAGE3]

AGENT CONTROLE

LOCAL CONSTRAINTS
CAPABILITY
O(Fermer_porte | ∃ e ∈ ETAGES : e.Etat_porte)
```

2. OBLOG.

2.1 INTRODUCTION.

OBLOG, qui est l'acronyme de « *OBject LOGic* », a été développé au Portugal par l'équipe du Professeur Sernadas dans le cadre du projet européen ESPRIT II/III IS-CORE (Information Systems - Correctness and Reusability) dans le but d'être utilisé lors de la phase de conception (ingénierie du logiciel). Ses concepteurs se sont rendus compte que cet outil pouvait être étendu au cycle de vie complet des systèmes d'information avec un grand avantage qui est la continuité entre les différentes phases du développement. Ensuite, est venue une phase d'industrialisation réalisée par le groupe Espírito Santo et Digital Equipment Europe. Actuellement, une des filiales du groupe Espírito Santo, OBLOG Software, poursuit la phase d'industrialisation et développe l'outil pour la plate-forme Windows.

OBLOG est un outil CASE composé d'un formalisme propre (le langage OBLOG), d'un environnement de développement (l'outil OBLOG) et d'une méthode de développement (ensemble de modèles, de concepts et de principes).

OBLOG est un langage graphique et formel. Grâce à ce formalisme, il est possible d'être précis dans les spécifications, de les vérifier d'un point de vue syntaxique et de les valider via des prototypes. OBLOG est aussi caractérisé par ses aspects orienté-objet et concurrent. Cependant, il est à noter que la concurrence se situe au niveau des objets et pas des actions propres à l'objet (pas de parallélisme entre les actions).

L'outil CASE OBLOG est composé d'un éditeur graphique orienté syntaxe[1], d'un checker, d'un générateur de code (non optimisé), d'un éditeur de boîtes de dialogue et permet la réalisation d'appel à du code C externe.

Les principes méthodologiques se basent sur l'utilisation d'un seul et même langage tout au long du cycle de développement du logiciel. Vu le caractère orienté-objet du langage, il est également possible d'emprunter les principes de génie logiciel OO. De plus, il ne faut pas perdre de vue qu'OBLOG a été développé dans le but de permettre la réutilisabilité et le travail coopératif. C'est pourquoi, il faut un environnement de développement supportant le travail de groupe. Cet environnement sera développé autour d'un « *repository* ».

[1] Il est à noter que l'outil OBLOG ne supporte qu'un sous-ensemble du langage proposé par l'équipe du Professeur Sernadas dans [SER94-2].

2.2 CONCEPTS DU LANGAGE.

2.2.1 GROUPE.

En OBLOG, le concept de groupe permet de classer les différents projets globaux de l'organisation par thème. Chaque groupe est composé d'une ou plusieurs unités. L'outil OBLOG développé pour cette division des unités est le « *Session Manager* » dont un exemple est donné à la Figure 2-1. Cet exemple contient un groupe (*Version1* marqué par un G), une unité (*Ascenseur* marqué par un U), des objets (marqués par un O) et des types de données (marqués par un D).

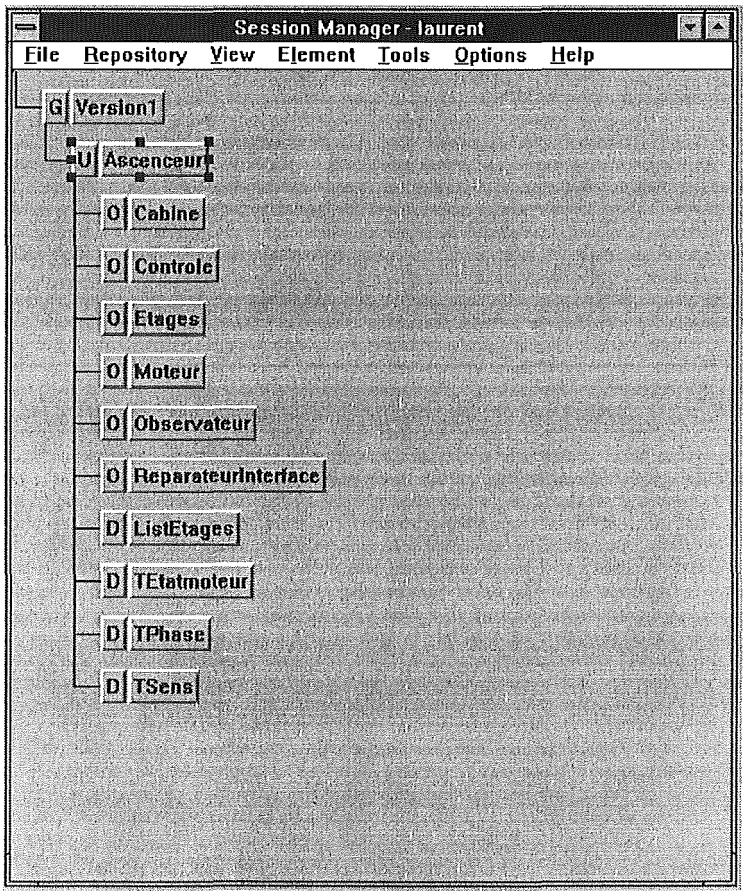


Figure 2-1 : Session Manager.

2.2.2 UNITÉ.

Les unités servent à faire le lien entre l'univers du discours et les différents objets en faisant partie. L'élément OBLOG qui permet ce lien est le « *Community Diagram* » présenté à la Figure 2-3. Chaque unité contient une partie des objets propres au projet global. Cette dé-

coupe permet de diviser le projet global en sous-projets pour ainsi distribuer les responsabilités entre les différents analystes et concepteurs.

2.2.3 OBJET.

Un objet est une entité qui encapsule des attributs caractérisant cette entité et des actions (ou méthodes) ayant un rapport avec celle-ci. Il est important de signaler qu'un objet OBLOG n'est pas l'équivalent d'un objet de la littérature OO puisque la visibilité est totale pour les attributs et les méthodes de tous les objets. Tout objet a une identité par un « *surrogate* » et est caractérisé par un état qui peut changer avec le temps. Il existe trois types d'objets en OBLOG : OBL,DBX et TBL. Le type OBL est le type par défaut d'un objet. Le type DBX signifie que l'objet sert d'interface avec l'utilisateur et le type TBL que l'objet permet de rendre ses données permanentes.

2.2.4 CLASSE D'OBJETS.

La classe d'objets (aussi appelée classe) sert à décrire une famille d'objets ayant des propriétés similaires, des comportements communs, des relations communes avec d'autres objets et une sémantique commune. La sémantique attachée à l'objet est importante parce que suivant celle-ci, deux objets peuvent dans une modélisation faire partie d'une même classe et dans une autre appartenir à des classes différentes. La Figure 2-2 vous montre la différence entre un objet et une classe d'objets.

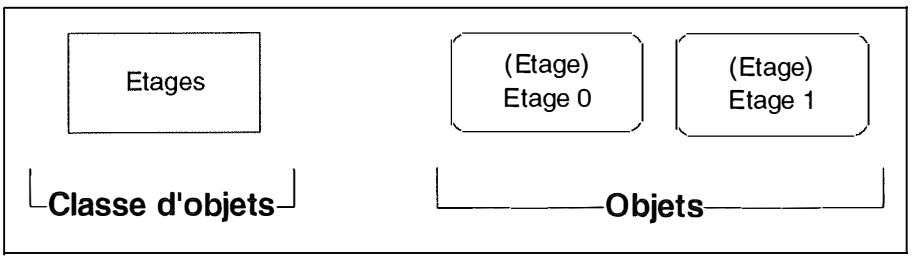


Figure 2-2 : Classe et objets.

2.2.5 HÉRITAGE.

L'héritage se définit comme étant une relation entre une classe et une sous-classe. Cette relation permet à la sous-classe d'utiliser les attributs et les méthodes (ou actions) définis dans la super-classe. La super-classe se définit comme une généralisation de ses différentes sous-classes et les sous-classes comme des spécialisations de la super-classe.

Dans OBLOG, la notion d'héritage est plus limitée que celle développée dans d'autres langages OO : seul le simple héritage a été défini et chaque graphe d'héritage n'est composé que d'objets ayant le même type (OBL, DBX et TBL). De plus, l'héritage ne porte que sur les attributs et les actions, pas sur le comportement (même si lors de la création du lien d'héritage, le graphe de comportement est copié dans l'objet qui hérite).

2.2.6 ATTRIBUT.

Les attributs sont les données associées à chaque objet, caractérisant les propriétés statiques (l'état) des objets.

2.2.7 ACTION.

L'action est le seul mécanisme qui permette le changement d'état d'un objet. Ces actions peuvent être soit actives, soit passives. Les actions passives ont besoin d'être déclenchées par un appel extérieur pour se réaliser alors que les actions actives se déclenchent de leur propre initiative. Dans les deux cas, la transition associée à l'action doit être la transition courante et les conditions associées à cette transition doivent être vérifiées. Une action est caractérisée par sa signature, les mises à jour des attributs de l'objet et les appels qu'elle réalise.

2.2.8 APPEL.

L'appel est le mécanisme qui permet la communication entre les objets. Il peut être réalisé par des actions actives ou passives mais n'est à destination que d'actions passives. Les appels peuvent avoir un seul destinataire (*single call*) ou plusieurs destinataires (*multi-call*).

Dans le cas d'un appel, deux cas se présentent : soit l'instance appelée est dans un état permettant de recevoir l'appel et à ce moment l'action appelée est réalisée, soit elle ne l'est pas et l'appel échoue. Nous présentons plus en détails les appels à la section Editeur d'appels (2.3.4).

2.2.9 COMPORTEMENT ADMISSIBLE.

Le comportement admissible définit l'enchaînement des actions pour un objet. Il est décrit par un graphe d'états (appelés également situations) et de transitions (« *Behaviour Diagram* »). A chaque transition est associée une action de l'objet. Dans ce graphe, les transitions peuvent être tirées pour autant que l'état courant soit l'état origine de la transition et que la précondition qui lui est associée soit vérifiée.

2.3 ÉLÉMENTS DE L'ÉDITEUR.

Dans cette section, nous ne décrivons que les éléments de l'outil CASE OBLOG pour Windows, à savoir le « *Community Diagram* », le « *Declaration Diagram* », le « *Behaviour Diagram* » et l'éditeur d'appel.

2.3.1 COMMUNITY DIAGRAM.

L'élément OBLOG dans lequel on déclare les classes d'objets, leurs interactions, leurs associations, les relations d'héritage, les types énumérés et les listes s'appelle le « *Community Diagram* ». Il vous est présenté à la Figure 2-3.

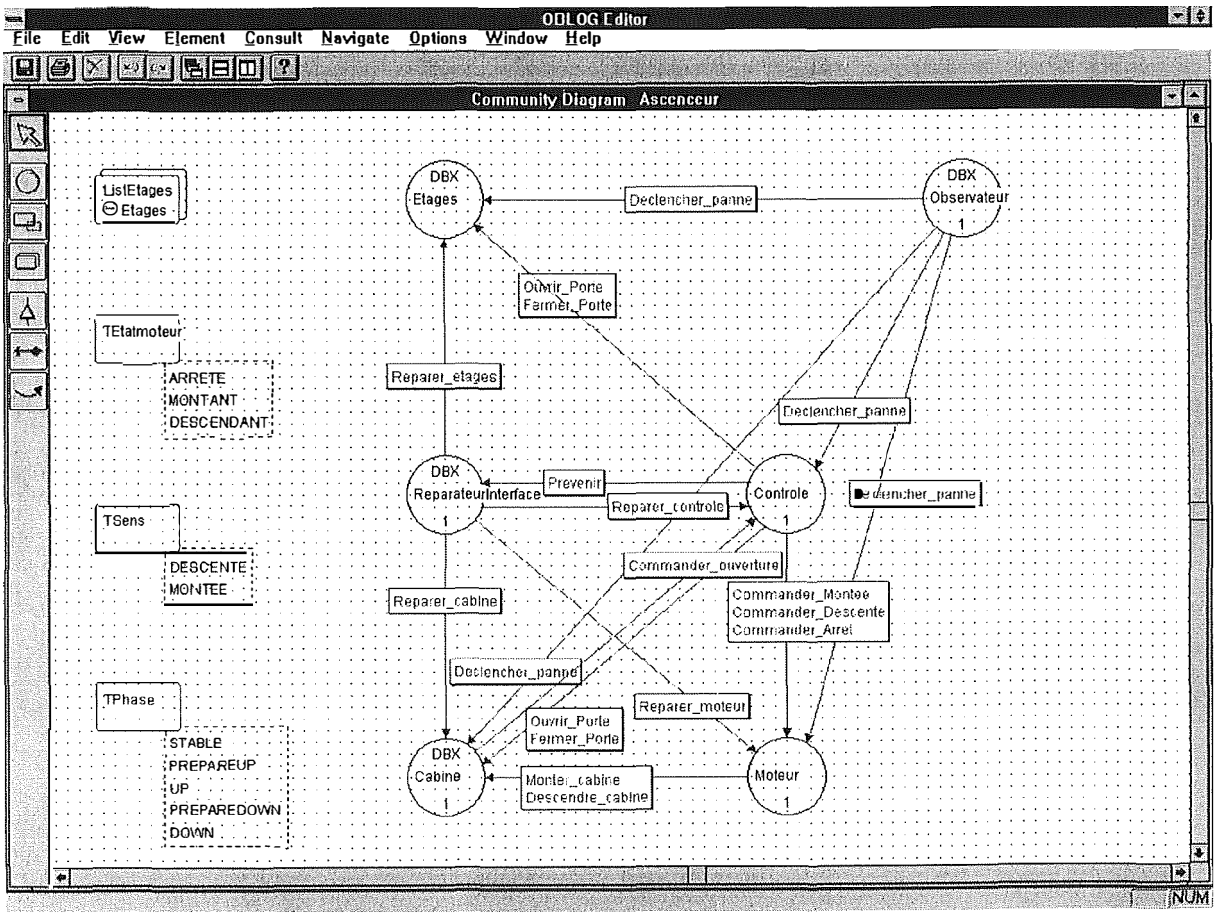


Figure 2-3 : Community Diagram.

Comme le montre cette figure :

- Les objets sont représentés par des cercles dans lesquels sont indiqués le nom de la classe, son type (OBL, DBX, TBL) et si elle est à instance unique (indiquée par le chiffre 1 à l'intérieur du cercle).

- Les types énumérés sont représentés par deux rectangles à bords arrondis légèrement superposés, un en trait plein, l'autre en traits discontinus. Le premier contient le nom du type et le second les différentes valeurs que celui-ci peut prendre.
- Les listes ont pour représentation deux rectangles à bords arrondis superposés avec le nom de la liste et son type.

La relation d'héritage est désignée par un trait entrecoupé par un triangle entre deux classes. La sous-classe est celle qui est du coté de la base du triangle. Un exemple est donné à la Figure 2-5 : Declaration Diagram.. Il faut lire ce graphe comme suit : Les classes *EtageBas*, *EtageInter* et *EtageHaut* sont des spécialisations de la classe abstraite *Etages*.

Les associations sont représentées par des traits liant les objets et la cardinalité peut être 0-1 (Figure 2-4 c), 1-1 (Figure 2-4 d), 0-M (Figure 2-4 a) ou 1-M (Figure 2-4 b). Les exemples suivant se lisent comme suit : à l'Objet 1 est associé de 0 à M Objet 2 (Figure 2-4 a), à l'Objet 1 est associé de 1 à M Objet 2 (Figure 2-4 b), ...Il est à rappeler que la cardinalité s'exprime dans les deux sens. Nous pouvons donc aussi dire d'après les exemples que, à tout Objet 2 est associé 0 ou 1 Objet 1.

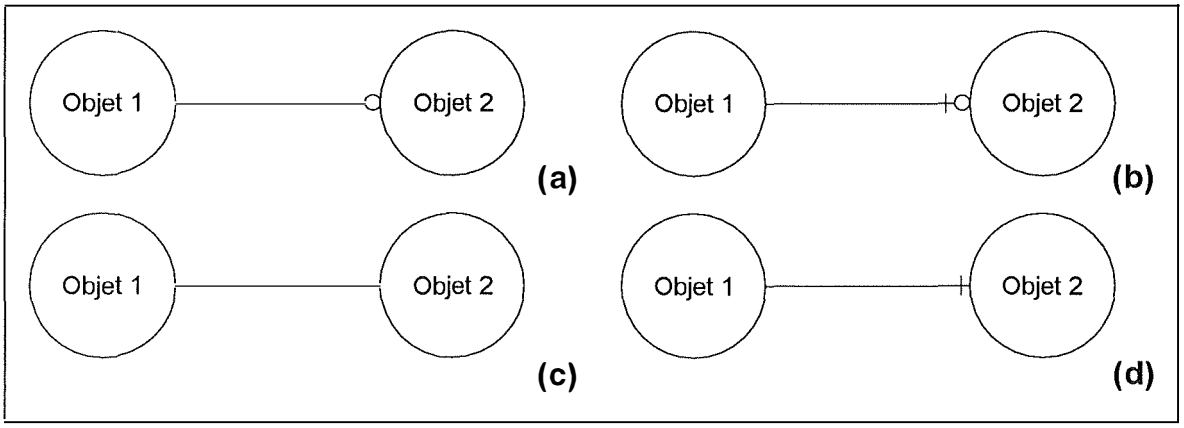


Figure 2-4 : Cardinalité des associations.

Les interactions sont représentées par des flèches sur lesquelles se fixent des rectangles contenant le nom des différentes interactions. Des exemples sont donnés à la Figure 2-3.

Les interactions et les associations ont pour seul but de représenter ces relations entre les objets. Elles n'ont aucun effet sur la spécification.

2.3.2 DECLARATION DIAGRAM.

Le « *Declaration Diagram* » est le composant OBLOG dans lequel on déclare la structure d'un objet c'est-à-dire ses attributs, ses actions et la signature de ces actions. Lorsqu'un objet hérite d'un autre, les attributs et les actions héritées non modifiés se situent à la

gauche du cercle représentant l'objet. Tandis que tout ce qui est propre à l'objet se situe à sa droite. Un exemple vous est présenté à la Figure 2-5.

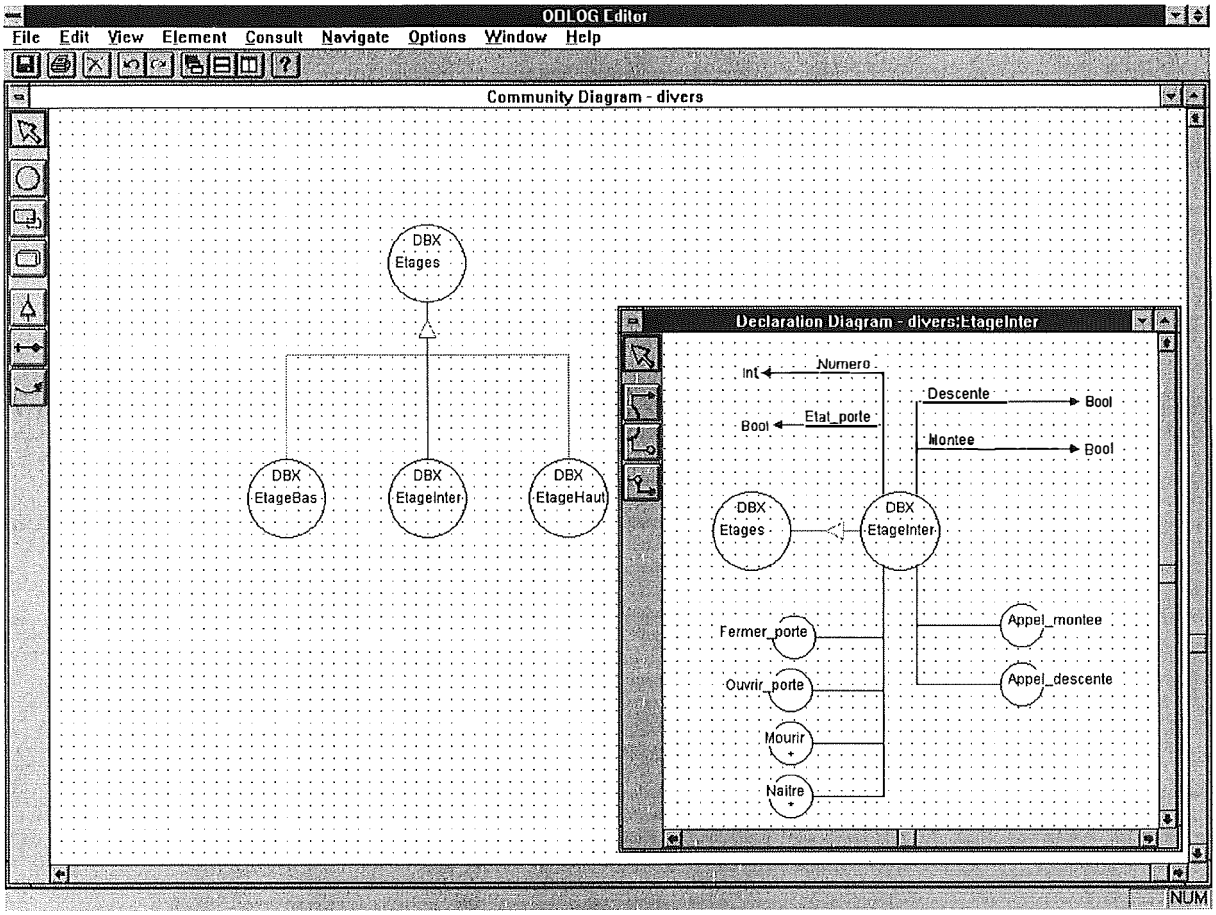


Figure 2-5 : Declaration Diagram.

2.3.2.1 Attributs.

Les attributs sont représentés par des flèches nommées pointant sur le codomaine et se situant au-dessus du cercle représentant l'objet. Ces attributs doivent être typés soit par des types simples soit par des types définis par l'utilisateur.

Il existe six types simples en OBLOG : *Boolean*, *Character*, *Natural*, *Integer*, *Real* ou *String*. Ces types sont prédéfinis et ne peuvent être modifiés.

Les types définis par l'utilisateur sont les types énumérés, les listes ou les classes d'objets. Tous ces types sont définis dans le « *Community Diagram* ».

2.3.2.2 Actions.

Les actions sont représentées par de petits cercles se situant au-dessous du cercle représentant l'objet et contenant le nom de celles-ci. Comme dit précédemment, il existe deux

types d'actions, les actions passives et les actions actives. Pour les différencier, les actions actives sont marquées par un point d'exclamation.

Les actions peuvent avoir des paramètres qui sont représentés par des flèches nommées pointant sur le codomaine du paramètre et ces flèches ont comme origine l'action à laquelle elles sont rattachées.

Il existe trois types d'actions :

- les actions de naissance qui servent à créer l'objet et à initialiser les attributs,
- les actions de mises à jour,
- les actions de mort qui servent à détruire l'objet.

2.3.3 BEHAVIOUR DIAGRAM.

Comme dit précédemment, le « *Behaviour Diagram* » sert à décrire le comportement d'un objet grâce à un graphe d'état transition. Un exemple est présenté à la Figure 2-6.

Les états aussi appelés situations sont nommés. La première situation du graphe de comportement est une situation de naissance et la dernière une situation de mort (il est à noter que celle-ci est facultative). Elles sont marquées respectivement d'une astérisque(*) et d'un plus(+) devant leur nom. Il peut y avoir plusieurs situations de naissance et de mort.

Une transition est composée d'une situation source, d'une situation destination et d'une action. A chaque transition peuvent être associées une condition (indiquée sur le graphe par un point d'interrogation) et une instanciation (indiquée par un :=) . La condition va contraindre la réalisation de l'action et l'instanciation permet de définir la valeur des paramètres de l'action. La première action doit être une action de naissance et la dernière une action de mort c'est-à-dire que les actions de naissance ont pour origine une situation de naissance et que les actions de mort ont pour destination une situation de mort.

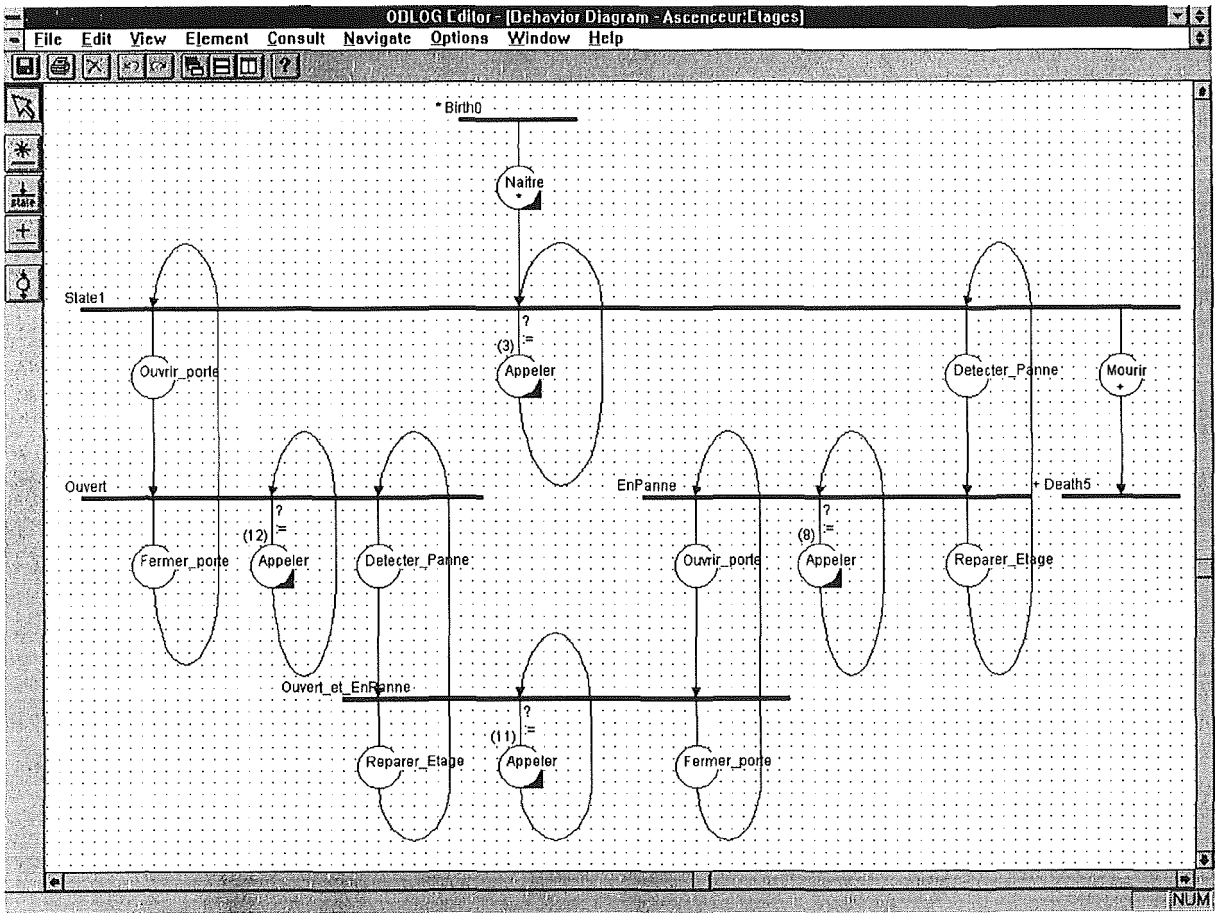


Figure 2-6 : Behaviour Diagram.

2.3.4 EDITEUR D'APPELS.

2.3.4.1 Appels en détail.

Comme expliqué dans la section concepts du langage (2.2), les appels sont le mécanisme mis en œuvre dans OBLOG pour permettre la communication entre les agents. Ils sont réalisés par des actions. Les paramètres de l'appel sont la classe d'objets que l'on appelle, la référence d'une instance ou d'une liste d'instances de l'objet appelé, l'action de la classe, une condition d'appel et une instanciation des arguments de l'action appelée si celle-ci en possède.

Une action peut réaliser plusieurs appels et ces appels peuvent être conditionnés. Si la condition de l'appel n'est pas vérifiée au moment de l'exécution de l'action, l'appel n'a pas lieu mais n'est pas considéré comme échouant et l'action est réalisée si aucun appel n'échoue.

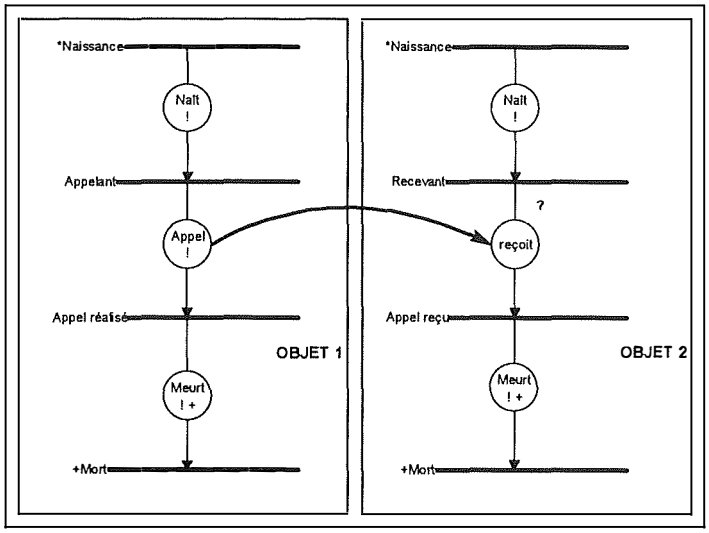


Figure 2-7 : Le mécanisme de l'appel .

La Figure 2-7 montre deux objets qui communiquent. Cet appel sera considéré comme réussi si premièrement la situation courante de l'objet appelé est *recevant* et si deuxièmement la précondition de réalisation de l'action *reçoit* est vérifiée au moment de l'appel. Dans tous les autres cas, l'appel est considéré comme ayant échoué et la transition n'est pas tirée. On peut généraliser ceci à une chaîne d'appels : supposons que l'action *reçoit* réalise à son tour un appel, les principes énoncés auparavant restent valables. A ce moment, si l'appel de *reçoit* échoue, nous considérons que tous les appels de la chaîne ont échoué et aucune transition n'est tirée.

Il est à remarquer qu'une instance d'un objet ne peut pas faire partie deux fois d'une chaîne d'appels. Ceci s'explique par le fait que lorsqu'une instance est en train de réaliser un appel, elle n'est pas en état d'en recevoir.

2.3.4.2 Description.

L'éditeur vous est présenté à Figure 2-8. Dans un premier écran, vous pouvez voir les différents appels réalisés par l'objet CONTROLE tel l'appel de l'action *Fermer_porte* vers l'action *Fermer_porte* de CABINE et de ETAGES. Dans un second écran, vous trouvez toutes les informations concernant l'appel : la classe cible, l'action cible, l'identifiant de l'instance, la condition d'appel et les instanciations des paramètres de l'action appelée.

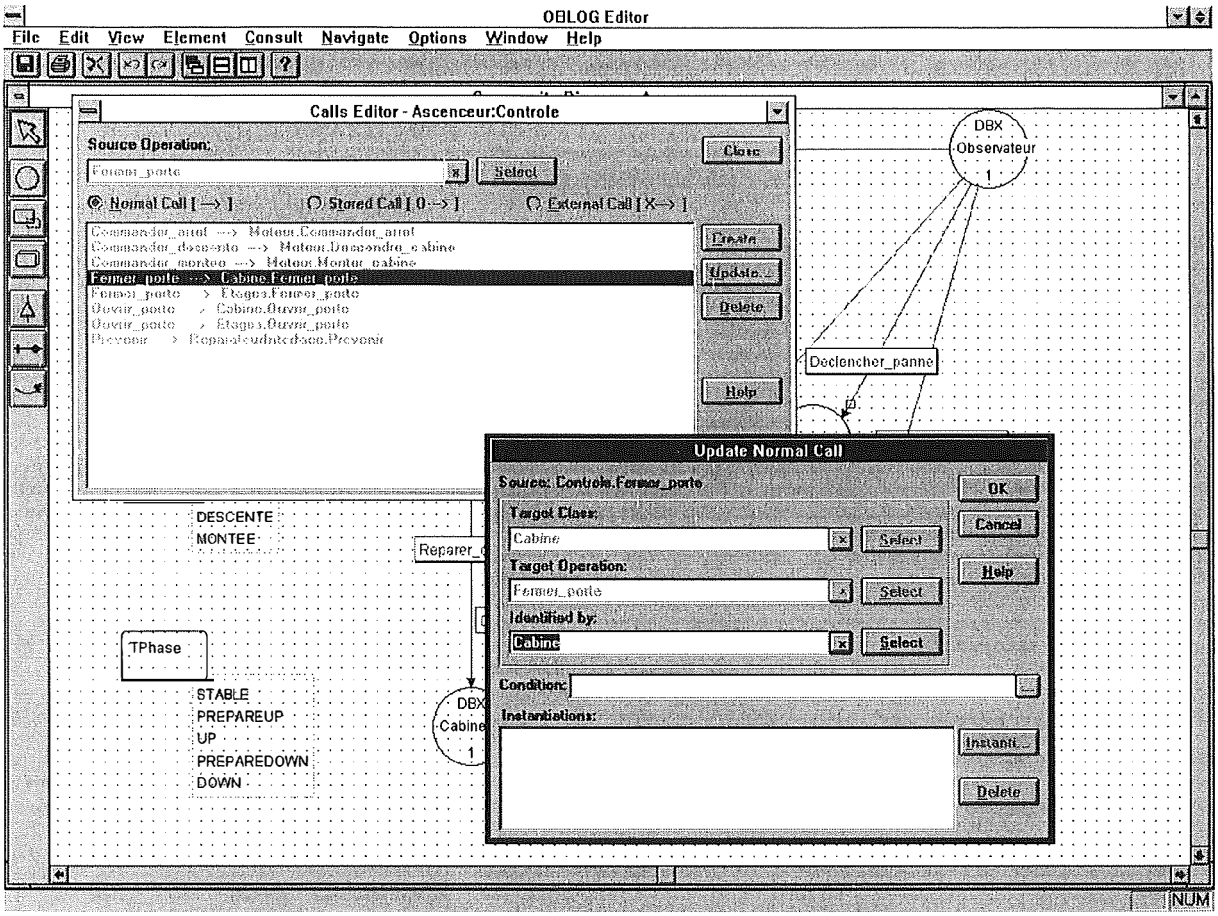


Figure 2-8 : Editeur d'appels.

2.4 MÉCANISME DE FONCTIONNEMENT.

2.4.1 CONCURRENCE ENTRE LES OBJETS.

Comme dit dans l'introduction, OBLOG est un langage concurrent au niveau de l'objet c'est-à-dire qu'il se partage une ressource, à savoir le temps d'exécution, puisque rien ne se fait en parallèle. Pour gérer ce temps de partage, OBLOG est équipé d'un « *scheduler* » de tâches qui donne la main aux différents objets de façon aléatoire. Lorsqu'un objet a la main, il peut réaliser une action. Les actions sont considérées comme atomiques c'est-à-dire qu'elles se déroulent toujours entièrement. Si l'action en cours de réalisation comporte des appels, on va considérer que l'ensemble des actions faisant partie de la chaîne d'appel ne forment qu'une seule et même action.

2.4.2 SÉQUENCE DE RÉALISATION D'UNE ACTION.

La séquence de réalisation d'une action est présentée à la Figure 2-9. Premièrement, il y a le déclenchement de l'action soit par un appel pour les actions passives soit de leur propre initiative pour les actions actives. A ce moment, les paramètres de l'action sont instanciés. Ensuite, il y a l'évaluation de la condition de transition en se basant sur les valeurs des attributs avant leur mise à jour et des paramètres. Si cette condition n'est pas vérifiée, la transition n'est pas tirée. S'il n'y pas d'appel à réaliser, la mise à jour des attributs est effectuée et la transition tirée. Par contre, dans le cas où il y aurait des appels, ceux-ci sont réalisés comme suit : Evaluation de la condition d'appel, instanciation des paramètres de l'action appelée et déclenchement de l'action appelée. Si l'appel en cours de traitement échoue, alors on arrête le tout et la transition n'est pas tirée. Par contre, si aucun appel n'échoue, la mise à jour des attributs est effectuée et la transition tirée.

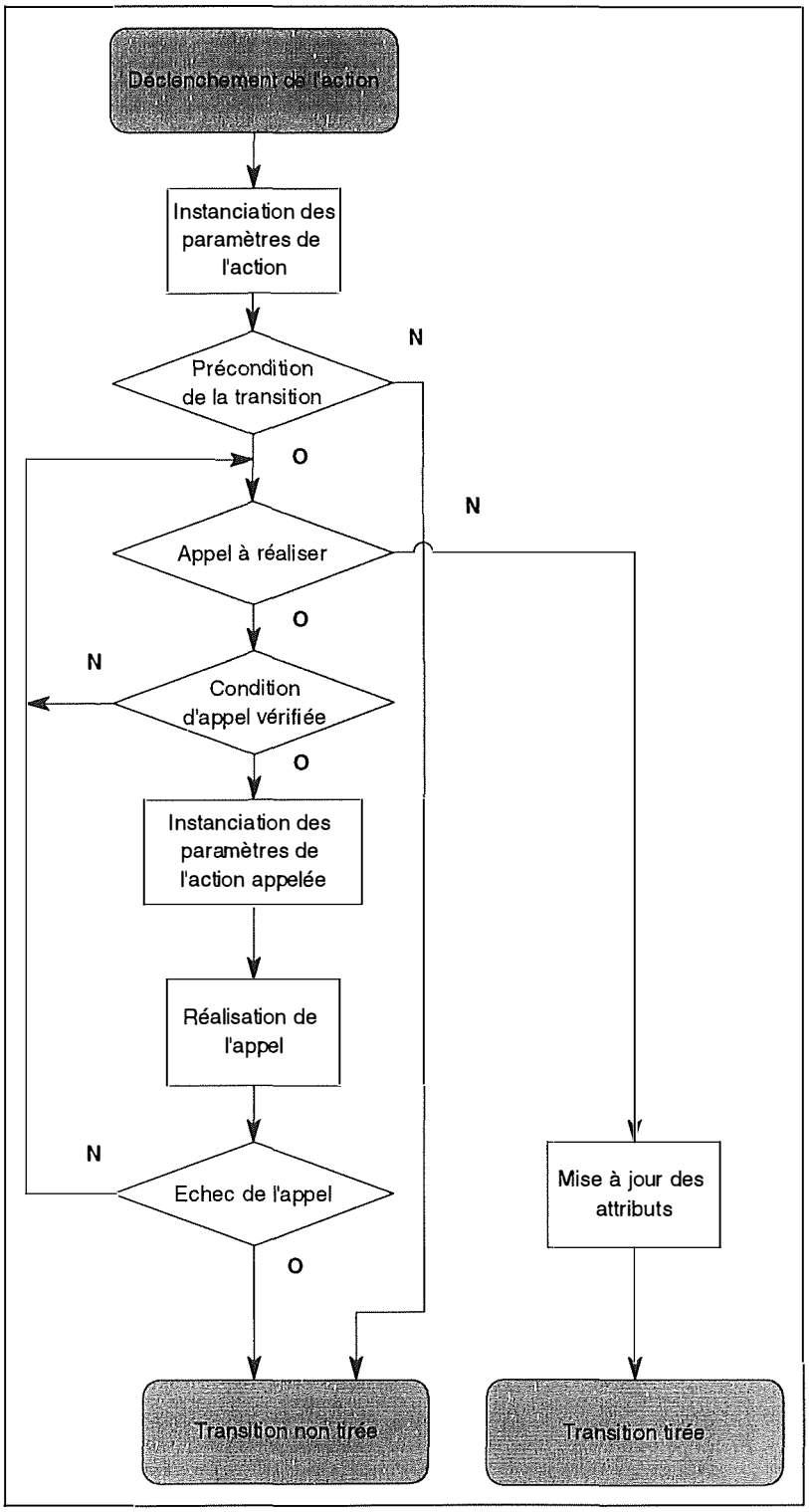


Figure 2-9 : Séquence de réalisation d'une action.

2.5 LE LANGAGE DES EXPRESSIONS.

Le langage OBLOG se base sur une logique typée du premier ordre, ceci afin d'offrir des possibilités de raisonnement. Nous vous présentons brièvement ce langage.

2.5.1 LA NAVIGATION DANS LES ATTRIBUTS.

Prenons l'exemple présenté à la Figure 2-10. Pour accéder à un des attributs de l'objet CONTROLE dans une expression, il suffit de faire suivre le nom *Controle* (nom de l'attribut de référence) par un "." et le nom de l'attribut désiré.

```
Appel_desc := if(Controle.Phase = TPhase$DOWN, FALSE, Appel_desc)
```

Cette assertion a pour signification que si la valeur de l'attribut *Phase* de l'objet CONTRÔLE identifié par *Contrôle* vaut DOWN (une des valeurs du type énuméré TPhase) alors *Appel_desc* est mis à FALSE sinon il reste inchangé.

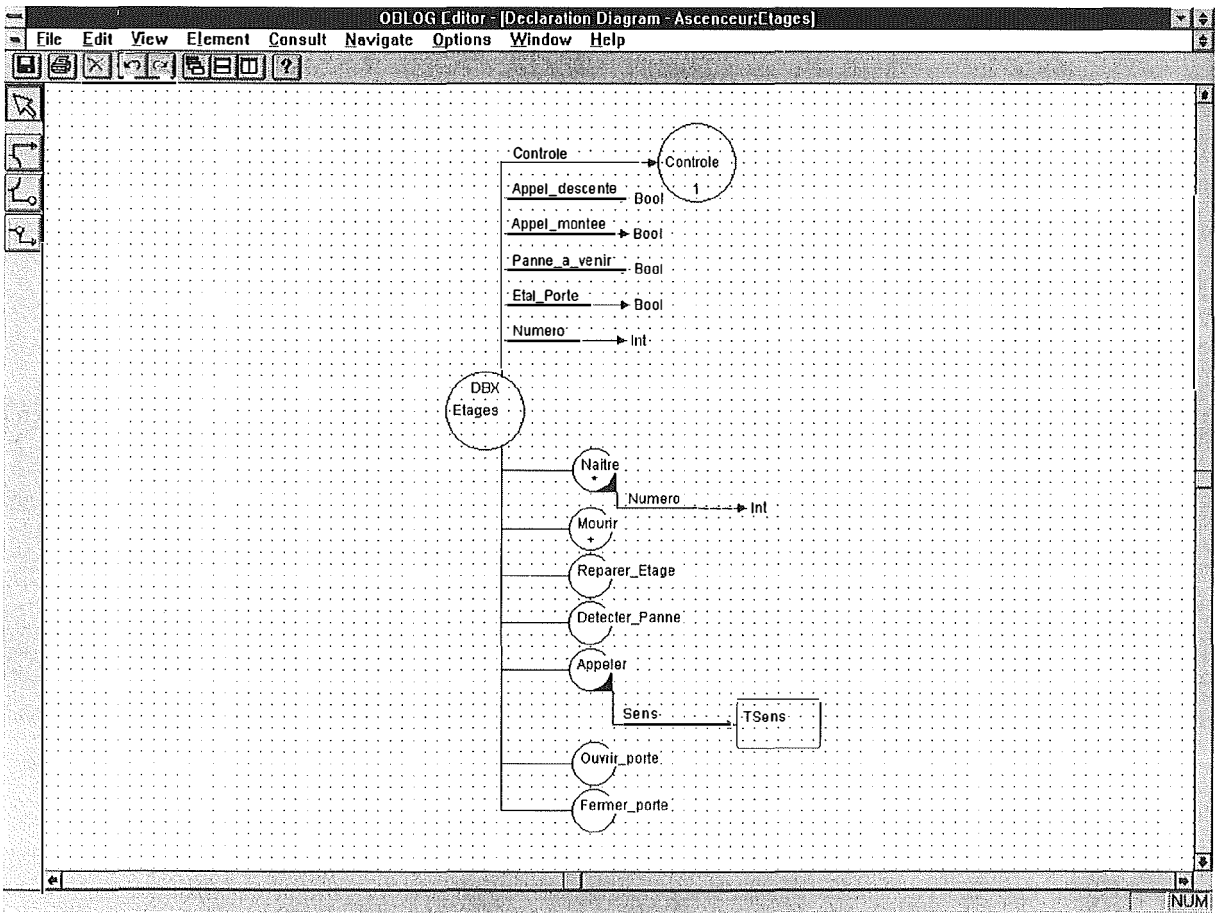


Figure 2-10 : Declaration Diagram.

2.5.2 LES TYPES ET LEURS OPERATIONS.

Il existe six types définis en OBLOG : Booléen, Entier, Rationnel, Naturel, Caractère et chaîne de caractères (string), auxquelles viennent s'ajouter les types énumérés, les listes ainsi que les objets. Les opérateurs classiques sur ces types sont également définis, vous les trouvez dans le Tableau 2-1.

Type	Opérateur	Type de retour
Booléen	AND	Booléen
Booléen	OR	Booléen
Booléen	NOT	Booléen
Entier, Rationnel, Naturel	+	Entier, Rationnel, Naturel
Entier, Rationnel, Naturel	-	Entier, Rationnel, Naturel
Entier, Rationnel, Naturel	*	Entier, Rationnel, Naturel
Entier, Rationnel, Naturel	/	Entier, Rationnel, Naturel
Entier, Naturel	% (modulo)	Entier, Naturel
String, Liste	‖ (concatenation)	String, Liste
String, Liste	INSERT(lst, nat,any)	String, Liste
String, Liste	APPEND(lst, any)	String, Liste
String, Liste	DELETE(lst, any)	String, Liste
String, Liste	REMOVE(lst, any)	String, Liste
Liste	FETCH(lst, nat)	Type de la liste
String, Liste, Enum	# lst	Naturel
Liste	ISIN(lst, any)	Booléen
Objet	EXISTS[NomObj | bool]	Booléen
Objet	ALL[NomObj | bool]	Liste d'objet
Objet	ONE[NomObj | bool]	Objet
/	IF(cond, val1, val2)	Type de val1 et val2

Tableau 2-1 : Opérateurs les plus couramment utilisés.

Cette liste d'opérateurs n'est pas exhaustive. A cela s'ajoutent les opérations de comparaison (=, <>, <, >, <= et >=) et les convertisseurs de types. Nous vous renvoyons à [OBL95-6] pour de plus amples détails.

La priorité sur ces différentes opérations est celle définie habituellement. Elle est ordonnée comme présenté dans le Tableau 2-2.

1	fonctions	5	<, >, <=, >=
2	- unaire, NOT, #	6	=, <>
3	*, /, %	7	AND
4	+, -, ‖	8	OR

Tableau 2-2 : Ordre de priorité des opérations.

2.5.3 LES EXPRESSIONS CONDITIONNELLES.

Il est habituel que le comportement d'une action puisse varier en fonction d'éléments extérieurs. Ceci s'exprime au niveau de la mise à jour des attributs grâce à l'opération IF. Son fonctionnement est le suivant : IF(condition, Valeur de retour si la condition est vérifiée, Valeur de retour si la condition n'est pas vérifiée). Cependant, un problème de représentation peut se présenter pour ce qui est des tables de décision. En effet, le nombre de IF imbriqués peut devenir très important et par là même la lisibilité quasi nulle .

2.5.4 LES REQUETES

On remarque dans le Tableau 2-1 la présence de trois opérations de requête (EXISTS, ALL, ONE).

1. L'opération EXISTS sert à savoir s'il existe une instance d'un certain objet satisfaisant une condition.
2. L'opération ALL sert à construire une liste contenant toutes les instances vérifiant la condition.
3. L'opération ONE renvoie une instance au hasard vérifiant la condition.

Dans les requêtes, deux mots réservés ont été ajoutés, Current et Self pour faire référence respectivement à l'instance en cours de traitement et à l'objet qui réalise la requête.

Exemples :

```
Etage :=   if(EXISTS[ Etages | Panne_a_venir ],
           ONE[ Etages | Panne_a_venir ], ONE[ Etages | TRUE ])
Position := ONE[ Etages | Numero = self.Position.Numero - 1 ]
Arret := EXISTS[ Etages | ISIN( self.Cabine.Etages_arret, current)
]
```

- La première assertion exprime une mise à jour conditionnelle. La condition porte sur l'existence d'un ETAGES dont l'attribut *Panne_a_venir* est à vrai. Si cette condition est vérifiée alors on sélectionne un ETAGES qui vérifie cette même condition sinon on en prend un au hasard.
- La seconde expression réalise la sélection d'un ETAGES dont l'attribut *Numero* est égal à l'attribut *Numero* de *Position* , qui est l'attribut de l'objet réalisant la requête, décrémenté de 1.
- La dernière expression renvoie vrai s'il existe un ETAGES se trouvant dans la liste *Etages_arret* de *Cabine,* attribut de l'objet réalisant cette requête.

2.6 COMMENTAIRES SUR LA SPECIFICATION.

Dans cette section, nous présentons différents commentaires sur l'architecture des objets ainsi que la description des attributs et des actions.

2.6.1 EXPLICATION DE L'ARCHITECTURE DES OBJETS.

2.6.1.1 Pour une simulation.

Le but dans une simulation est de développer un modèle collant au mieux avec la réalité pour pouvoir étudier le comportement du système. Ces modèles s'intéressent essentiellement à l'occurrence aléatoire des actions. Dans notre étude d'ascenseur, les occurrences d'actions des agents externes auraient pu avoir lieu en suivant des lois probabilistes. Nous aurions eu comme architecture, six objets propres au système (CONTRÔLE, MOTEUR, CABINE, REPARATEUR, ETAGES et UTILISATEURS) et un objet OBSERVATEUR. Le comportement des objets « externes » REPARATEUR et UTILISATEURS aurait dû être implémenté en C pour générer les variables aléatoires et les occurrences d'événements.

2.6.1.2 Pour un prototype.

Dans le cas d'un prototype, l'approche est différente puisque ce que nous cherchons à réaliser, c'est une maquette de ce que sera l'implémentation finale du système. De plus, le prototype devra servir pour la validation de la spécification. C'est pourquoi remplacer le comportement aléatoire des agents externes par des lois probabilistes n'a pas d'intérêt. Ce qu'il faut montrer au client, c'est que le prototype réagit adéquatement aux différents événements. Nous avons donc choisi de permettre à l'utilisateur de lui-même déclencher ces événements en supprimant les agents externes (leur rôle est pris en charge par l'utilisateur), en redistribuant les responsabilités des actions de ces agents entre les autres composants du système et en créant un objet interface servant à ces interactions.

Nous avons donc comme architecture quatre objets propres au système (CONTRÔLE, MOTEUR, CABINE et ETAGES), un objet OBSERVATEUR et un objet pour les interactions avec l'utilisateur (REPARATEURINTERFACE).

2.6.2 DESCRIPTION DES ATTRIBUTS.

2.6.2.1 Objet CONTROLE .

`Phase : TPHASE`

Attribut associé à *Phase* de la spécification ALBERT.

`Panne_a_venir : BOOL`

Attribut associé à *Panne_a_venir* de la spécification ALBERT.

`Prevenu : BOOL`

Attribut associé à *Prevenu* de la spécification ALBERT.

`Reparation : REPARATEURINTERFACE`

Artefact introduit dans la spécification pour permettre les interactions avec REPARATEUR.

`Moteur : MOTEUR`

Artefact introduit dans la spécification pour permettre les interactions avec MOTEUR et la lisibilité des attributs de MOTEUR.

`Cabine : CABINE`

Artefact introduit dans la spécification pour permettre les interactions avec CABINE et la lisibilité des attributs de CABINE.

2.6.2.2 Objet MOTEUR.

`Etat : TETATMOTEUR`

Attribut associé à *Etat* de la spécification ALBERT.

`Panne_a_venir : BOOL`

Attribut associé à *Panne_a_venir* de la spécification ALBERT.

`Cabine : CABINE`

Artefact introduit dans la spécification pour permettre les interactions avec CABINE.

2.6.2.3 Objet CABINE.

`Etat_porte : BOOL`

Attribut associé à *Etat_porte* de la spécification ALBERT.

`Position : ETAGES`

Attribut associé à *Position* de la spécification ALBERT.

`Panne_a_venir : BOOL`

Attribut associé à *Panne_a_venir* de la spécification ALBERT.

`Etages_arret : LISTETAGES`

Attribut associé à *Etages_arret* de la spécification ALBERT.

`Moteur : MOTEUR`

Artefact introduit dans la spécification pour permettre les interactions avec MOTEUR et la lisibilité des attributs de MOTEUR.

`Controle : CONTROLE`

Artefact introduit dans la spécification pour permettre les interactions avec CONTROLE et la lisibilité des attributs de CONTROLE.

`Numero_dest : INT`

Artefact introduit dans la spécification pour la saisie de l'étage de destination.

2.6.2.4 Objet ETAGES.

`Numero : INT`

Attribut associé à *Numero* de la spécification ALBERT.

`Etat_porte : BOOL`

Attribut associé à *Etat_porte* de la spécification ALBERT.

`Appel_montee : BOOL`

Attribut associé à *Appel_montee* de la spécification ALBERT.

`Appel_descente : BOOL`

Attribut associé à *Appel_descente* de la spécification ALBERT.

`Panne_a_venir` **`instance of`** `BOOLEAN`

Attribut associé à *Panne_a_venir* de la spécification ALBERT.

`Controle : CONTROLE`

Artefact introduit dans la spécification pour permettre les interactions avec CONTROLE et la lisibilité des attributs de CONTROLE.

`Sens : String`

Artefact introduit dans la spécification pour la saisie du sens de l'appel.

2.6.2.5 Objet REPARATEURINTERFACE.

`Controle : CONTROLE`

Artefact introduit dans la spécification pour permettre les interactions avec CONTROLE et la lisibilité des attributs de CONTROLE.

`Moteur : MOTEUR`

Artefact introduit dans la spécification pour permettre les interactions avec MOTEUR et la lisibilité des attributs de MOTEUR.

`Cabine : CABINE`

Artefact introduit dans la spécification pour permettre les interactions avec CABINE et la lisibilité des attributs de CABINE.

`Etage : ETAGES`

Artefact introduit dans la spécification pour permettre les interactions avec ETAGES et la lisibilité des attributs de ETAGES.

`NbrEtages : INT`

Artefact introduit dans la spécification pour la création des ETAGES.

2.6.2.6 Objet OBSERVATEUR.

Tous les attributs qui ne seront pas décrits dans cette section n'ont pour raison d'exister que leur utilité dans l'affichage d'informations.

`NoEtage : INT`

Artefact introduit pour le rafraîchissement des attributs des différentes instances d'ETAGES.

`Etage : ETAGES`

Artefact introduit dans la spécification pour permettre la lisibilité des attributs de ETAGES.

`PositionEd : INT`

Artefact introduit pour l'affichage du numéro de l'étage où se trouve la cabine.

`Reparateur : REPARATEURINTERFACE`

Artefact introduit dans la spécification pour permettre la création de l'objet REPARATEURINTERFACE.

`Cabine : CABINE`

Artefact introduit dans la spécification pour permettre les interactions avec CABINE et la lisibilité des attributs de CABINE.

`Moteur : MOTEUR`

Artefact introduit dans la spécification pour permettre les interactions avec MOTEUR et la lisibilité des attributs de MOTEUR.

`Contrôle : CONTROLE`

Artefact introduit dans la spécification pour permettre les interactions avec CONTROLE et la lisibilité des attributs de CONTROLE.

2.6.3 DESCRIPTION DES ACTIONS.

2.6.3.1 Objet CONTROLE

`Naitre`

Artefact introduit pour la création de l'objet.

`Mourir`

Artefact introduit pour la destruction de l'objet.

`Attendre_5s`

Action active de temporisation pour permettre la sortie des utilisateurs de la cabine.

`Attendre_5s_passif`

Action passive de temporisation pour permettre la sortie des utilisateurs de la cabine.

```
Changer_Phase
     NouvellePhase : TPHASE
```

Artefact introduit dans la spécification pour gérer les mouvements de la cabine.

`Commander_arret`

Action commandant l'arrêt du moteur.

`Commander_descente`

Action commandant la mise en route du moteur pour descendre la cabine.

`Commander_montee`

Action commandant la mise en route du moteur pour monter la cabine.

`Detecter_panne`

Action de détection d'une panne potentielle.

```
Fermer_porte
     Etage : ETAGES
```

Action commandant l'ouverture des portes de la cabine et de l'étage où se trouve la cabine.

```
Ouvrir_porte
     Etage : ETAGES
```

Action commandant la fermeture des portes de la cabine et de l'étage où se trouve la cabine.

`Prevenir`

Action prévenant le réparateur de la présence de pannes potentielles.

`Reparer_Controle`

Artefact introduit dans la spécification pour la mise à jour de l'attribut *Panne_a_venir*.

`Referencer`

Artefact introduit dans la spécification pour obtenir les références des objets avec lesquels il y a des interactions ou des accès aux attributs.

`VerifierExistence`

Artefact introduit dans la spécification pour obtenir les références des objets avec lesquels il y a des interactions ou des accès aux attributs.

`Commander_ouverture`

Artefact introduit dans la spécification pour le contrôle de l'occurrence d'actions (*Ouvrir_porte*, ...).

2.6.3.2 Objet MOTEUR

`Naitre`

Artefact introduit pour la création de l'objet.

`Mourir`

Artefact introduit pour la destruction de l'objet.

`*Monter_cabine`

Action de montée de la cabine.

`*Descendre_cabine`

Action de descente de la cabine.

`Detecter_panne`

Action de détection d'une panne potentielle.

`Reparer_Moteur`

Artefact introduit dans la spécification pour la mise à jour de l'attribut *Panne_a_venir*.

`Commander_arret`

Artefact introduit dans la spécification pour la mise à jour de l'attribut *Etat*.

2.6.3.3 Objet CABINE

`Naitre`

Artefact introduit pour la création de l'objet.

`Mourir`

Artefact introduit pour la destruction de l'objet.

`Detecter_panne`

Action de détection d'une panne potentielle.

`Reparer_Cabine`

Artefact introduit dans la spécification pour la mise à jour de l'attribut *Panne_a_venir*.

`Ouvrir_porte`

Artefact introduit pour la mise à jour de l'attribut *Etat_porte*.

`Fermer_porte`

Artefact introduit pour la mise à jour de l'attribut *Etat_porte*.

`Commander_ouverture`

Action externe venant du fait de la suppression de l'agent UTILISATEURS.

`Choisir_destination`

Action externe venant du fait de la suppression de l'agent UTILISATEURS.

`Monter_Cabine`

Artefact introduit pour la mise à jour de l'attribut *Position*.

`Descendre_Cabine`

Artefact introduit pour la mise à jour de l'attribut *Position.*

`Referencer`

Artefact introduit dans la spécification pour obtenir les références des objets avec lesquels il y a des interactions ou des accès aux attributs.

`VerifierExistence`

Artefact introduit dans la spécification pour obtenir les références des objets avec lesquels il y a des interactions ou des accès aux attributs.

2.6.3.4 Objet REPARATEUR

`Naitre`

Artefact introduit pour la création de l'objet.

`Mourir`

Artefact introduit pour la destruction de l'objet.

`Reparer_contrôle`

Action de réparation de l'élément contrôle.

`Reparer_moteur`

Action de réparation de l'élément moteur.

`Reparer_cabine`

Action de réparation de l'élément cabine.

```
Reparer_etages
     Etages : LISTETAGES
```

Action de réparation des éléments étages.

`Prevenir`

Artefact introduit en contrepartie de *Prevenir* de CONTROLE.

`Creation_controle`

Artefact introduit pour déclencher la naissance de l'objet CONTROLE.

`Creation_moteur`

Artefact introduit pour déclencher la naissance de l'objet MOTEUR.

`Creation_cabine`

Artefact introduit pour déclencher la naissance de l'objet CABINE.

`Creation_etage`

Artefact introduit pour déclencher la naissance d'un objet ETAGES.

2.6.3.5 Objet ETAGES

`Naitre`

Artefact introduit pour la création de l'objet.

`Mourir`

Artefact introduit pour la destruction de l'objet.

`Detecter_panne`

Action de détection d'une panne potentielle.

`Reparer_Moteur`

Artefact introduit dans la spécification pour la mise à jour de l'attribut *Panne_a_venir*.

`Ouvrir_porte`

Artefact introduit pour la mise à jour de l'attribut *Etat_porte*.

`Fermer_porte`

Artefact introduit pour la mise à jour de l'attribut *Etat_porte*.

```
Appeler
     Sens : TSENS
```

Action externe venant du fait de la suppression de l'agent UTILISATEURS.

2.6.3.6 Objet OBSERVATEUR.

`Naitre`

Artefact introduit pour la création de l'objet.

`Mourir`

Artefact introduit pour la destruction de l'objet.

`Referencer`

Artefact introduit dans la spécification pour obtenir les références des objets avec lesquels il y a des interactions ou des accès aux attributs.

`VerifierExistence`

Artefact introduit dans la spécification pour obtenir les références des objets avec lesquels il y a des interactions ou des accès aux attributs.

`Rafraichir`

Artefact introduit dans la spécification pour mettre à jour les attributs d'affichage.

`RafraichirEtage`

Artefact introduit dans la spécification pour mettre à jour les attributs d'affichage.

`SelectionEtage`

Artefact introduit dans la spécification pour mettre à jour les attributs d'affichage.

2.7 COMMENTAIRES SUR LE LANGAGE.

Nous avons remarqué lors du développement d'applications en OBLOG différents détails et problèmes que nous allons commenter.

2.7.1 PAUVRETE EN TERME DE CONSTRUCTEURS DE TYPES.

Les seules structures de données complexes existant en OBLOG sont les listes et les types énumérés. Il est vrai qu'il est possible de simuler quelques-uns des autres types généralement utilisés. Mais il serait quand même plus pratique de doter ce langage de constructeurs pour les ensembles, les multi-ensembles, les tables et les unions.

2.7.2 ABSENCE D'OPERATIONS SUR LES TYPES CONSTRUITS.

Il est bien souvent intéressant de pouvoir définir des opérations sur les types construits. Or, le langage OBLOG ne le permet pas.

2.7.3 PRESENCE D'ATTRIBUTS ARTEFACTS.

La première remarque vient de la présence d'attributs artefacts pour permettre la communication entre les objets. Il est vrai que certains d'entre eux sont nécessaires pour identifier les objets destinataires de l'appel mais lorsque nous réalisons un appel vers à objet à instance unique, faut-il encore l'identifier ? Ceci a pour conséquence d'éloigner la spécification de l'univers du discours et de rendre la spécification difficilement compréhensible par le client.

2.7.4 GESTION DE L'INDETERMINISME.

La gestion de l'indéterminisme est réalisée au niveau du « *scheduler* » de tâches, c'est lui qui choisit l'objet qui aura la main ainsi qu'une des transitions potentiellement réalisables. Seulement, si pour une situation donnée, il existe plusieurs transitions pouvant être tirées, le « *scheduler* » choisira toujours la même. Il est cependant impossible de prédire laquelle. Donc, nous pouvons dire qu'il y a une forme d'indéterminisme même si ce n'est pas de l'indéterminisme total.

2.7.5 DISCONTINUITE ENTRE LE COMMUNITY DIAGRAM ET LE DECLARATION DIAGRAM.

Lors de la phase de conception, avec l'aide d'OBLOG, il est possible de décrire l'architecture objet, les associations entre ces objets et leurs interactions. Malheureusement,

lorsque nous passons à la phase de description des ces différents objets, seuls les héritages sont pris en compte. En effet, le squelette de l'objet (ses attributs et ses actions) ne comporte que les éléments hérités. Or, il serait intéressant d'avoir pour chaque association un attribut (qui pourrait être modifié par la suite) représentant cette association. Il en est de même pour les interactions qui peuvent se concrétiser que par des actions. Il serait donc utile de les incorporer dans le squelette de l'objet.

3. ETUDE DE PROTOTYPAGE

3.1 INTRODUCTION.

3.1.1 LE MODELE DU « WATERFALL ».

Le modèle du « *Waterfall* » a été présenté dans [ROY70] et [BOE81]. Nous voyons à la Figure 3-1 qu'il est composé de 4 phases principales : La phase de *spécification*, la phase de *design* aussi appelée la phase de *conception*, la phase d'*implémentation* et la phase d'*opération et maintenance*.

3.1.1.1 La spécification.

La phase de spécification comprend une étude de faisabilité et la production de documents, appelés habituellement cahiers des charges, spécifiant les fonctionnalités attendues du système informatique décrit, l'environnement, la plate-forme de travail, les langages de programmation, les interfaces, les performances désirées, ...

3.1.1.2 La conception.

La production de la phase de conception comprend une description de l'architecture *software* du système organisée autour du concept de module. Cette architecture comprendra des modules décrivant le traitement, les interfaces, les données persistantes, le *middleware* et le système d'exploitation.

3.1.1.3 L'implémentation.

La phase d'implémentation comprend l'écriture du code des différents composants du programme, leur intégration ainsi que leur implémentation. L'intégration consiste à regrouper les différents composants et à les faire fonctionner ensemble. L'implémentation consiste à rendre opérationnel le produit de l'intégration. Ceci peut demander l'installation du programme, la conversion des données des versions précédentes et la formation des utilisateurs.

3.1.1.4 L'opération et la maintenance.

Lors de la phase d'opération et de maintenance, les utilisateurs peuvent se servir du programme tout en ayant ponctuellement des mises à jour software.

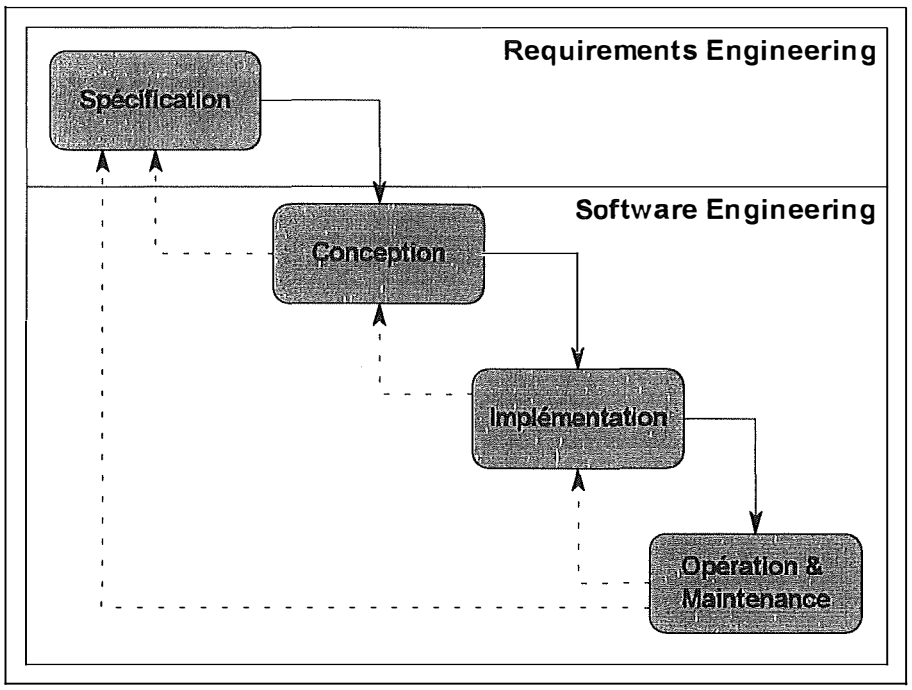


Figure 3-1 : Le modèle du "Waterfall"

3.1.2 LE PROTOTYPAGE DANS LE MODELE DU « WATERFALL ».

Suite à l'application de ce modèle ou d'une de ses versions dérivées, on a remarqué que le produit final, c'est-à-dire le programme, ne rencontrait pas toujours les besoins effectifs des clients. Pour éviter ce genre de mésaventure et parce que bien souvent les spécifications formelles ne sont que peu compréhensibles par l'homme de la rue, on a décidé d'ajouter aux différentes phases des étapes de vérification et de validation. La vérification sert à savoir si ce que la phase a produit, a bien été fait. Tandis que la validation sert à savoir si, ce qui a été produit, est bien ce qui était désiré. Pour faciliter la validation, les analystes et les concepteurs peuvent se baser sur des prototypes. Ceci permet de présenter aux clients des prototypes afin de tenir compte de leur avis lors de chaque phase de développement. Si les clients réagissent positivement à ce prototype, le développement se poursuit. Par contre, si le prototype ne correspond pas à leurs attentes , il faut apporter des corrections. Il est intéressant d'attirer l'attention sur les raisons de la non correspondance aux attentes des clients : soit les productions de la phase ne sont pas correctes, soit la réalisation du prototype a introduit des erreurs. Il est donc important de valider au mieux la phase de génération des prototypes. Il existe deux autres techniques comportementales de validation : la simulation et l'animation.

Le but premier du prototypage [HAB90] est de produire un programme exécutable à partir de spécifications formelles non-exécutables. Une fois terminé, le prototype pourra servir

pour valider un comportement concret du système avec des données concrètes. En d'autres mots, le prototype reproduit le comportement du système futur.

La simulation est une technique utilisée lorsque la spécification a un aspect non-déterministe, à savoir que l'occurrence d'actions n'est pas prédictible. Cette technique associera des lois probabilistes avec les occurrences d'actions.

L'animation sert à vérifier si un scénario présenté par l'utilisateur est valide pour le système spécifié. C'est à l'utilisateur qu'incombe de lever l'indéterminisme en définissant le moment des occurrences des actions.

Comme nous venons de le voir, les prototypes permettent de développer dans un laps de temps court des programmes exécutables, la contrepartie étant le peu d'efficacité de ces prototypes et le fait qu'ils ne répondent pas à toutes les spécifications. Bien souvent les concepteurs de prototypes devront mettre de côté une partie des spécifications.

La Figure 3-2 présente le modèle du « *Waterfall* » enrichi de la phase de prototypage au niveau de la phase de spécification.

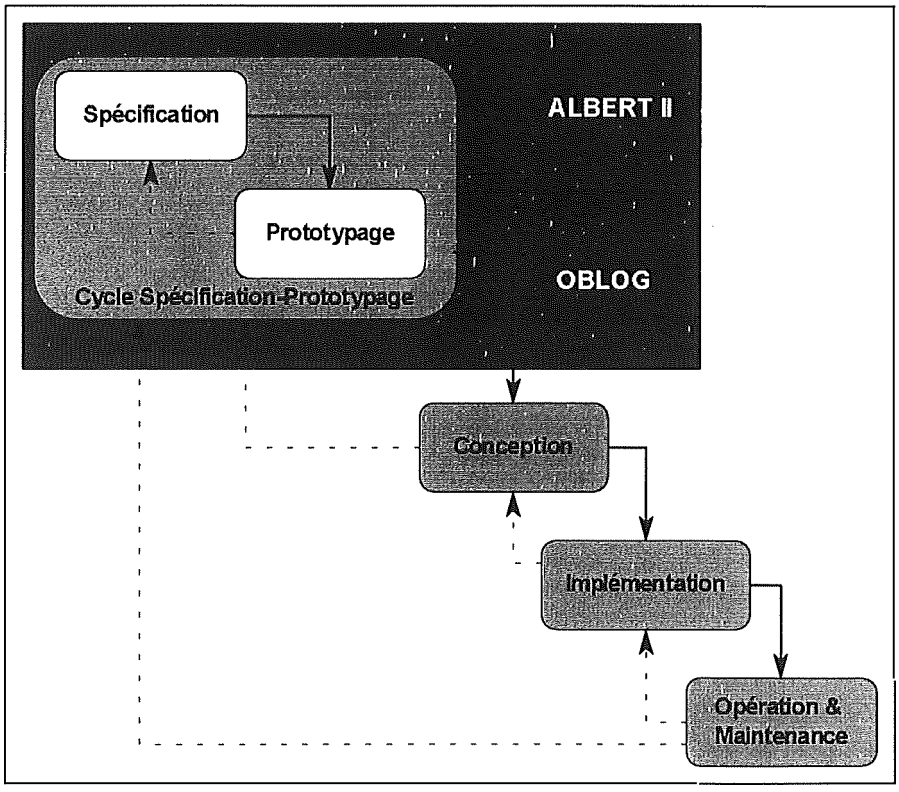


Figure 3-2 : Le prototypage dans le modèle du « Waterfall ».

3.1.3 UTILISATION D'ALBERT II ET D'OBLOG.

Dans la Figure 3-2, nous montrons le lien entre les phases de développement d'un logiciel et les langages ALBERT II et OBLOG. Nous savons qu'ALBERT II est un langage de

spécification destiné à l'ingénierie des besoins et qu'OBLOG, même si ses concepteurs espèrent étendre son utilisation à tout le cycle de vie du logiciel, est destiné à l'ingénierie du logiciel (phase de conception et d'implémentation automatique). C'est d'ailleurs pourquoi OBLOG est utilisé pour réaliser des prototypes. Malheureusement, il y a un prix à payer, puisqu'OBLOG ne permet pas de tenir compte des contraintes temporelles. Nous devrons donc faire abstraction du temps pour le prototype.

3.2 METHODOLOGIE POUR GENERER LE PROTOTYPE.

Nous présentons la méthodologie de façon générale, elle sera vue plus en détail ultérieurement. Cette méthodologie correspond à un pseudo-algorithme de traduction.

1. Transformation des types de données prédéfinis.
2. Suppression des agents externes.
   - 2.1. Déplacement des fonctionnalités vers d'autres agents.
   - 2.2. Création d'interfaces pour gérer les fonctionnalités non déplacées.
3. Suppression de l'indéterminisme dans les agents restants.
4. Identification des agents qui affichent ou saisissent des informations.
5. Transformation des agents en objets.
   - 5.1. Traduction des composants d'état en attributs.
   - 5.2. Traduction des actions.
6. Ajout des actions de naissance et de mort.
7. Construction de l'objet OBSERVATEUR.
8. Construction des graphes de comportement.
   - 8.1. Traduction locale.
   - 8.2. Intégration locale.
   - 8.3. Intégration globale.
9. Création des appels.

3.2.1 TRANSFORMATION DES TYPES DE DONNEES PREDEFINIS.

Cette transformation de types consiste à remplacer les types d'ALBERT II par leur équivalent OBLOG lorsque ces types existent. Le Tableau 3-1 vous présente ces correspondances.

Type ALBERT II	Type OBLOG
Booléen	Booléen
Entier	Entier, Naturel

Rationnel	Rationnel
Caractère	Caractère
String	String
Enum	Enum
Ensemble	Liste
Séquence	Liste
Multi-ensemble	Liste
Table	Liste
Union	-

Tableau 3-1 : Correspondance des types.

Nous proposons pour les types dont la transformation n'est pas évidente les solutions suivantes :

3.2.1.1 Ensemble.

En fait, la liste en OBLOG peut être utilisée comme un ensemble puisqu'il existe des opérations d'inclusion, d'appartenance et de retrait d'un élément. La seule précaution à prendre est d'éviter les doublons. Pour cela, il existe une opération sur les listes qui suppriment ces doublons. Nous vous renvoyons à [OBL95-6].

3.2.1.2 Multi-ensemble.

Les remarques faites pour l'ensemble restent valables mais dans ce cas, il n'est plus nécessaire de gérer les doublons.

3.2.1.3 Table.

Pour ce qui est des tables, il est possible de créer leur équivalent grâce à deux listes : une liste d'entrée, servant à déterminer la position de l'élément correspondant, et une liste de sortie contenant les correspondants. Il faudra bien évidemment traiter ces listes en parallèle.

3.2.2 LES OPERATIONS.

Il n'y aucun moyen en OBLOG de construire des opérations sur les types définis par l'utilisateur. Il n'est donc pas possible de traduire ces opérations directement.

3.2.3 SUPPRESSION DES AGENTS EXTERNES.

Le but du prototype étant de montrer aux clients ce à quoi va ressembler l'application finale, il n'est pas nécessaire de maintenir les agents faisant partie de l'environnement du système dans la spécification puisque leur comportement sera piloté par l'utilisateur du prototype. Cependant, il faudra soit déplacer les interactions entre l'environnement et le système vers d'autres agents soit créer des interfaces pour les gérer. Ceci a pour but de permettre à l'utilisateur de déclencher les événements à comportement aléatoire venant des agents de l'environnement.

Le déplacement des interactions vers d'autres agents a pour effet de forcer le type de l'objet qui les reçoit. En effet, ces objets hôtes se voient affecter des fonctionnalités d'interface avec l'utilisateur. Ils devront donc être de type DBX.

3.2.4 SUPPRESSION DE L'INDETERMINISME.

Il est tout d'abord nécessaire de préciser de quel type d'indéterminisme nous parlons. Nous nous concentrons essentiellement sur le caractère purement aléatoire des occurrences d'actions (tel que les phénomènes de pannes). Cependant derrière les contraintes de temps se dissimule également de l'indéterminisme comme on le voit dans l'exemple suivant :.

Exemple :

```
STATE COMPONENTS
Comp instance of BOOLEAN
ACTION
Maj
LOCAL CONSTRAINTS
STATE BEHAVIOUR
¬Comp ⇒◊<5' Comp
EFFECTS OF ACTIONS
Maj : Comp := True
```

Cet exemple exprime la contrainte selon laquelle le composant d'état *Comp* doit avoir la valeur TRUE dans les 5 minutes suivant son changement de valeur à FALSE. Nous avons bien un comportement indéterministe mais celui-ci ne nous intéresse pas.

Pour supprimer l'indéterminisme qui nous intéresse, nous proposons que ce soit l'utilisateur qui déclenche ces événements. Nous allons donc déplacer le déclenchement de ces

actions vers un autre objet que nous appellerons OBSERVATEUR qui servira d'interface avec l'utilisateur du prototype. Nous dédoublerons donc l'action avec sa partie déclencheur dans l'objet OBSERVATEUR et sa partie réactive dans l'objet d'origine.

3.2.5 IDENTIFICATION DES AGENTS QUI AFFICHENT OU SAISISSENT DES INFORMATIONS.

L'identification de ces agents va servir à la phase suivante pour le choix des types des objets. Tout agent qui devra afficher des données ou en saisir auprès de l'utilisateur sera représenté dans la spécification OBLOG par un objet DBX.

3.2.6 TRANSFORMATION DES AGENTS EN OBJETS.

Pour chaque agent restant, nous créerons un objet dont le type sera soit DBX, soit OBL suivant le résultat de la phase précédente et le nombre d'instances possibles sera le même que dans la spécification ALBERT . Pour ce qui est de l'objet servant à lever l'indéterminisme (OBSERVATEUR), le nombre d'instances maximum sera un. Les objets remplaçant les agents externes auront autant d'instances que ces agents externes en avaient.

Par défaut, nous ferons correspondre à tous les composants d'état un attribut. Mais cette règle n'est pas applicable si, dans la spécification ALBERT II, des composants d'état ont été introduits pour des raisons de contrôle d'occurrences d'actions. Dans ce cas, ces artefacts n'auront aucun équivalent en OBLOG.

Nous appliquons la même règle pour les actions. A toute action ALBERT correspond une action OBLOG sauf pour les actions qui servent à nommer les « *Action Composition* ». De plus, vu qu'en OBLOG, il n'est pas possible de modifier les valeurs d'un attribut autrement que par une action propre à l'objet, il faut pour chaque « *Effect of Action* » portant sur des actions externes dupliquer l'action. Il ne faudra pas oublier que l'occurrence d'une action ayant des effets sur un autre agent devra donner lieu à un appel en OBLOG. Il faudra également dédoubler les actions importées se trouvant dans l' « *Action Composition* » même si celles-ci n'ont pas d'« *Effet of Action* ». En effet, leur présence sert à déclencher une « suite » d'actions. Elles donneront également lieu à un appel.

3.2.7 AJOUT DES ACTIONS DE NAISSANCE ET DE MORT.

Comme tout objet OBLOG est composé d'une action de naissance et d'une action de mort, nous ajoutons ces actions. De plus, pour ce qui est de l'« *Initial Valuation* », nous pouvons réaliser son équivalent dans l'action de naissance puisqu'elle sera la première action

réalisée. Nous proposons que toutes les actions de naissances soient actives et nous les ferons suivre par une action dont le but est d'obtenir les références des instances existantes. Il faut cependant s'assurer que toutes les instances ont déjà été créées lors de cette obtention. Vous pouvez trouver un exemple dans le graphe de comportement de l'objet CABINE en annexe.

3.2.8 CONSTRUCTION DE L'OBJET OBSERVATEUR.

La seule raison d'exister de cet objet est de permettre à l'utilisateur de pouvoir visualiser les différents attributs des objets ainsi que de pouvoir déclencher les événements indéterministes. Cet objet comportera donc autant d'attributs qu'il y en a dans la spécification. De plus, il aura comme actions les doubles déclencheurs des actions indéterministes du système. Il faudra qu'il soit également équipé de fonctions de rafraîchissement des attributs.

Pour ce qui est de l'intégration de l'objet OBSERVATEUR, elle est assez simple puisque cet objet ne reçoit aucun appel. Par contre, toutes les actions indéterministes dont il aura héritées seront accompagnées par un appel à l'action correspondant dans l'objet d'origine. Il aura également comme attributs des références vers toutes les instances du système pour pouvoir obtenir les valeurs des différents attributs de ces instances.

3.2.9 TRADUCTION LOCALE.

3.2.9.1 Pour un style « *State Behaviour* ».

Il existe deux types de contraintes exprimées dans le « *State Behaviour* » : les invariants et les contraintes d'évolution. Les contraintes qui conditionnent l'occurrence des actions sont du second type. Elles sont souvent de la forme *Antécédent* $\Rightarrow$ *ConnecteurTemporel Conséquent.*

La solution que nous proposons pour cette traduction se base sur les sous-graphes. En effet, en garantissant que dans le sous-graphe (composé de l'ensemble des situations qui peuvent être atteintes à partir de la situation d'origine à la transition associée à l'action rendant l'antécédent vrai), il n'y ait pas d'action de mort et que toutes les transitions dont la destination est la situation origine sont associées à une action rendant le conséquent vrai, nous garantissons également que la contrainte est vérifiée.

Voyons plutôt un exemple à la Figure 3-3 :

```
STATE COMPONENTS

Panne instance of boolean

ACTION

Declencher_panne
Reparer
...
STATE BEHAVIOUR

Panne ⇒ ◊¬ Panne

EFFECTS OF ACTION

Declencher_panne : Panne := TRUE
Reparer : Panne := FALSE
```

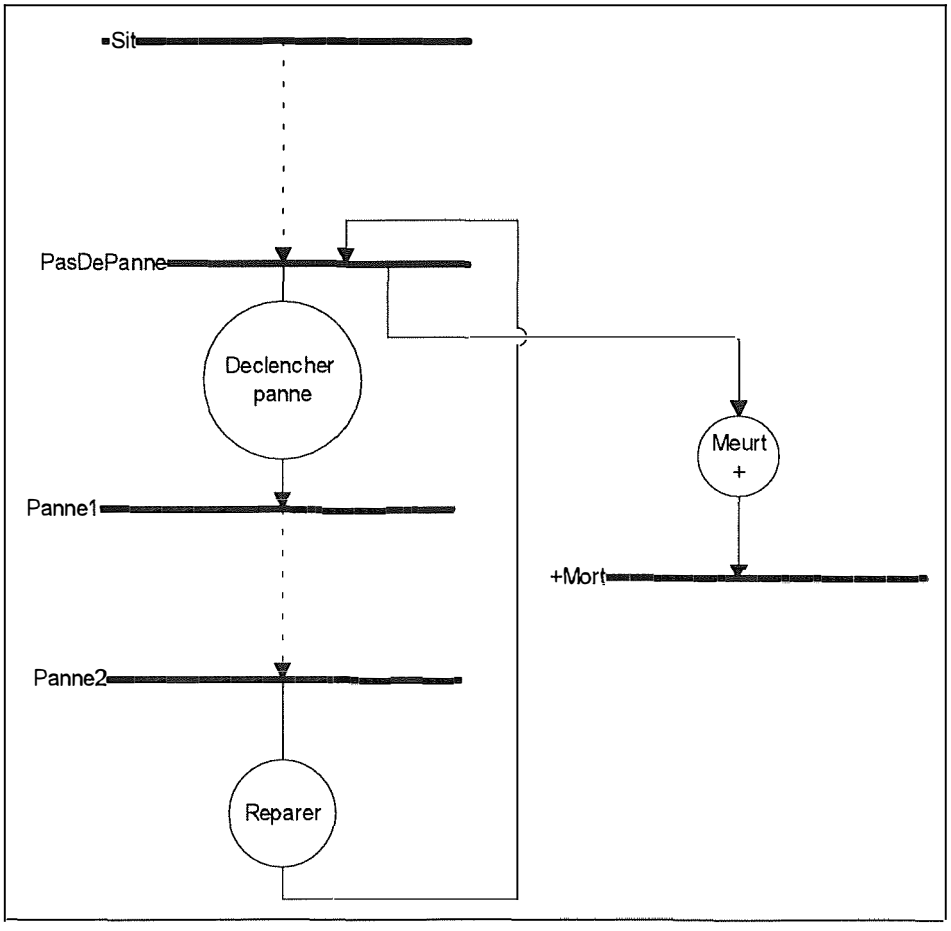


Figure 3-3 : Behaviour Diagram pour traiter les contraintes d'évolution.

Dans cet exemple, il faut garantir que si la situation courante est *PasDePanne* alors l'attribut *Panne* vaut False. Donc, toutes les transitions ayant pour destination PasDePanne devront avoir pour effet de mettre l'attribut *Panne1* à False. Dans ce cas, si un objet termine sa

vie, et s'il est passé par la situation *Panne* (où l'attribut *Panne* vaut True) alors il est passé après par la situation PasDePanne et la contrainte d'évolution est ainsi vérifiée.

3.2.9.2 Pour un style « *Etat - Transition* ».

Il existe trois sortes de contraintes lorsqu'on utilise le style « *Etat - Transition* » : l'obligation, l'interdiction et l'obligation exclusive. Nous proposons des solutions de traduction pour chacun de ces styles.

3.2.9.2.1 Obligation.

Les actions sur lesquelles il y a ce type de contraintes se traduiront en OBLOG par des actions actives. Ceci est dû au fait qu'en ALBERT si la condition est vérifiée alors l'action doit impérativement avoir lieu et ce à tout moment de la vie de l'agent. Or, cette action est interne et le seul moyen de forcer son occurrence est qu'elle soit active. Cependant, il ne faut pas perdre de vue que cette contrainte exprime le fait que l'action peut avoir lieu même si la condition n'est pas vérifiée.

Une version passive de l'action peut coexister si celle-ci est exportée vers un agent externe. Elle sera alors déclenchée par un appel.

Voyons un exemple :

```
STATE COMPONENTS

Cond1 instance of Boolean
Cond2 instance of Boolean

ACTION

Action1
Action2

CAPABILITY

O(Action1 | Cond1)
O(Action2 | Cond2)
```

Nous pouvons voir sur cet exemple que si la condition *Cond1* est vérifiée, seules les actions *Action1* et *Action12* peuvent être tirées et que si les conditions *Cond1* et *Cond2* ne sont pas vérifiées, toutes les transitions peuvent être tirées.

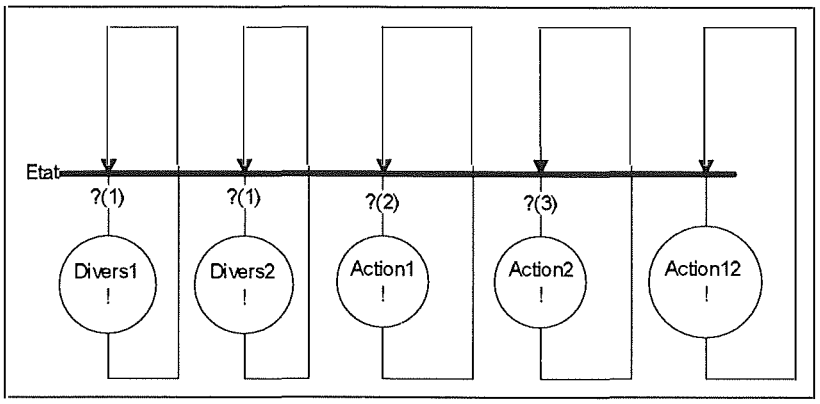


Figure 3-4 : Obligation.

```
?(1) : NOT Cond1 AND NOT Cond2
?(2) : NOT Cond2
?(3) : NOT Cond1
```

L'action *Action12* est équivalente à *Action1* || *Action2*.

Ce que nous proposons comme solution pour modéliser un tel comportement, c'est de placer à chaque situation (différente de celle de vie et de mort) l'action qui doit se produire et de poser sur toutes les autres transitions la condition exprimant la négation de celle forçant l'occurrence de cette action.

Un problème apparaît avec cette solution : s'il y a plusieurs actions qui sont sous le coup d'une telle contrainte, les conditions n'étant pas exclusives, la réalisation des actions peut se passer en parallèle . Or, ceci n'est pas possible en OBLOG. Nous proposons de modifier les conditions pour les rendre exclusives et de créer des actions supplémentaires qui ont le même résultat que la réalisation d'actions en parallèle.

Exemple :

O(Action1 | Cond1)	→	O(Action1 | Cond1 ∧ ¬¬Cond2)
O(Action2 | Cond2)	→	O(Action2 | ¬¬Cond1 ∧ Cond2)
	→	O(Action12 | Cond1 ∧ Cond2)

L'action *Action12* a le même effet que l'occurrence des actions *Action1* et *Action2* en parallèle.

Cette solution a comme inconvénient majeur l'explosion du nombre de transitions : si le nombre de conditions vaut n, le nombre de conditions exclusives vaudra n^2 -1 puisqu'il ne faut pas tenir compte du cas où toutes les conditions sont niées.

3.2.9.2.2 Interdiction.

Les actions sur lesquelles porte ce type de contraintes se traduiront soit par une action active, soit par une action passive suivant qu'elles sont déclenchées par l'agent lui-même ou par un autre agent. La contrainte d'interdiction se traduira par une précondition sur chacune des transitions associées à l'action. Par défaut, on permettra à l'action de se produire à chaque transition. Il se peut cependant que le comportement de l'action soit contraint par d'autres mécanismes. Il faudra bien évidemment en tenir compte.

Voyons un exemple :

```
STATE COMPONENTS
Cond1 instance of Boolean
ACTION
Action
CAPABILITY
I(Action | Cond1)
```

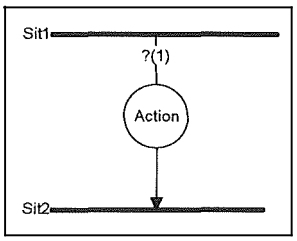


Figure 3-5 : Interdiction.

```
?(1) = Not Cond1
```

3.2.9.2.3 Obligation exclusive.

Les actions sur lesquelles il y a ce type de contraintes se traduiront également par des actions actives.

Voyons un exemple :

```
STATE COMPONENTS
Cond instance of Boolean
ACTION
Action
CAPABILITY
XO(Cond | Action)
```

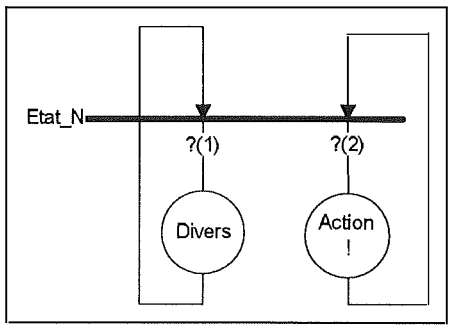


Figure 3-6 : Obligation exclusive.

Nous proposons comme solution pour modéliser un tel comportement de placer à chaque situation (différente de celle de vie et de mort) l'action qui doit se produire et de poser sur cette transition la condition équivalente à celle exprimée en ALBERT. Cependant, il faudra également inhiber le tirage des autres transitions à l'aide de conditions exprimant la négation de celle posée précédemment.

Le même problème que pour les contraintes d'obligation apparaît. Nous proposons la même solution pour y remédier : transformer les contraintes en contraintes exclusives et créer des actions équipotentes à la réalisation d'actions en parallèle.

3.2.9.3 Pour un style « *Action Composition* ».

Pour ce qui est de la traduction du style « *Action Composition* », la typologie est bien plus complexe. En effet, les « *Action Composition* » sont composées d'actions internes mais également d'actions externes. Or, comme nous l'avons dit dans la description d'OBLOG, l'appel est le seul mécanisme existant de communication. Donc, non seulement nous allons décrire le graphe de comportement mais aussi les appels permettant cette communication.

3.2.9.3.1 « *Action Composition* » composée d'actions internes.

3.2.9.3.1.1 Séquence.

Nous traduirons la séquence d'actions comme présentée dans l'exemple de la Figure 3-7.

En ALBERT II :

```
Action ↔ Action1 ; Action2 ; Action3
```

En OBLOG :

Dans ce cas-ci, *Action1, Action2* et *Action3* pourront se déclencher grâce à une contrainte d'obligation, une contrainte d'obligation exclusive, une contrainte d'évolution ou simplement d'elle-même. Ces trois actions seront donc actives.

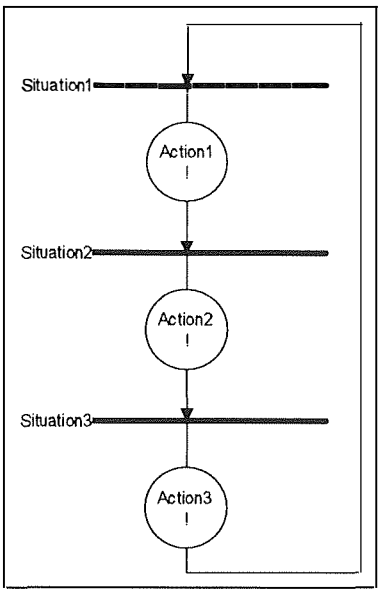


Figure 3-7 : Séquence d'actions internes.

3.2.9.3.1.2 Séquence de n occurrences.

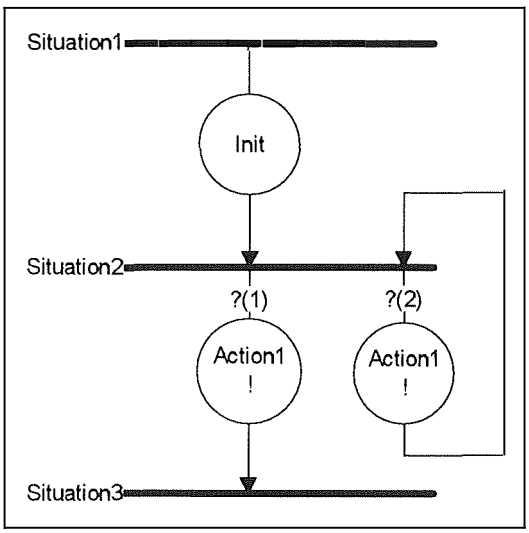
Nous traduirons la répétition séquentielle d'une action comme présentée dans l'exemple de la Figure 3-8.

En ALBERT II :

```
Action ↔ {Action1}^n
```

En OBLOG :

L'action *Init* sera déclenchée comme si elle avait été l'action *Action1*.




```
Operation Init
Updates
      i := 1

Operation Action1
Updates
      i := i + 1
      ...

?(1) i = n
?(2) i < n
```

Figure 3-8 : Séquence répétitive d'une action interne.

3.2.9.3.1.3 Conjonction.

La définition du langage ALBERT II nous garantit que si plusieurs actions se déroulent en même temps, elles n'ont pas d'effets contradictoires sur les composants d'état. Nous pouvons donc, pour reproduire le comportement d'une conjonction en ALBERT II, créer une seule action dont les effets seront équivalents à ceux de la réalisation des actions en parallèle.

En ALBERT :

```
Action ↔ Action1 ⊗ Action2 ⊗ Action3
```

En OBLOG :

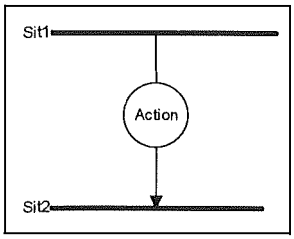


Figure 3-9 : Conjonction.

3.2.9.3.1.4 Conjonction parallèle.

La solution que nous proposons pour la conjonction parallèle est présentée à la Figure 3-10. Cette solution ne couvre qu'en partie la conjonction parallèle d'ALBERT II. En effet,

dans la solution présentée, les actions ne se passent pas en parallèle, il n'y a que des séquences. Il faudrait en plus introduire des actions dont les effets sont les mêmes que la réalisation d'actions en parallèle. Or, nous remarquons que le nombre de transitions est déjà important. Si, en plus, il fallait ajouter de nouvelles actions et les combinaisons de transitions s'y rapportant le graphe exploserait littéralement.

En ALBERT II :

```
Action ↔ Action1 || Action2 || Action3
```

En OBLOG :

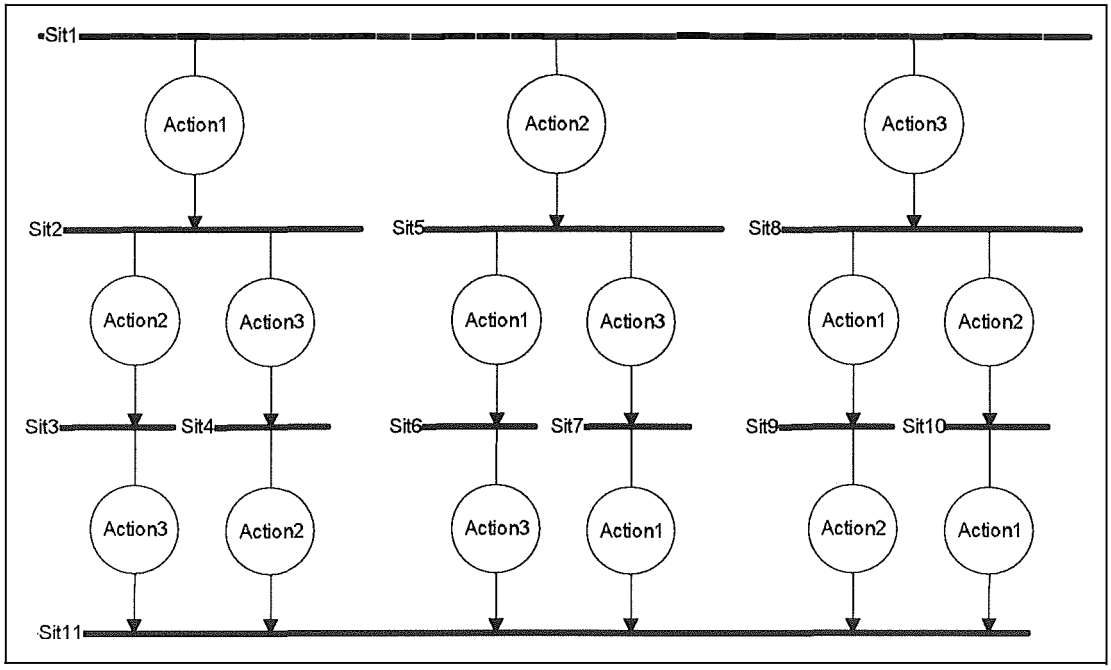


Figure 3-10 : Traitement de la conjonction parallèle.

Il est intéressant de savoir le nombre de transitions qu'il faudra construire pour modéliser un tel comportement. Nous vous présentons au Tableau 3-2 le nombre de transitions par rapport au nombre d'actions pour quelques valeurs. Il faut se rendre à l'évidence que modéliser un tel comportement devient vite un luxe (d'autant plus que l'implémentation du *scheduler* de tâches d'OBLOG n'a pas été réalisée avec le souci d'introduire un mécanisme gérant l'indéterminisme c'est-à-dire que, dans la version actuelle d'OBLOG, le *scheduler* choisit une transition, on ne sait pas dire laquelle mais on sait qu'il prendra toujours celle-ci pour une situation donnée).

Nombre d'actions	Nombre de transitions
2	4
3	15
4	64
5	325
6	1956

Tableau 3-2 : Ordre de grandeur du nombre de transitions.

3.2.9.3.1.5 Disjonction exclusive.

Ce type de comportement est aisément modélisable en OBLOG. Nous vous présentons une solution à la Figure 3-11.

En ALBERT II :

```
Action ↔ Action1 ⊕ Action2 ⊕ Action3
```

En OBLOG :

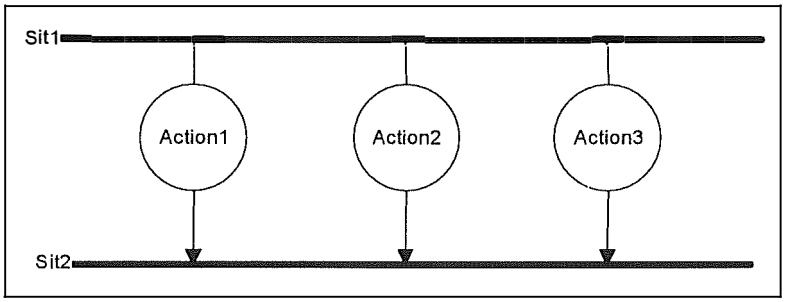


Figure 3-11 : Disjonction.

3.2.9.3.1.6 Dummy Action.

La DAC sert en ALBERT II à rendre une action potentiellement réalisable. Ceci peut être réalisé en OBLOG grâce à une action *Dummy* qui aura pour caractéristique de ne mettre à jour aucun attribut et de ne réaliser aucun appel.

En ALBERT II :

```
Action ↔ Action1 ⊕ Action2 ⊕ DAC
```

En OBLOG :

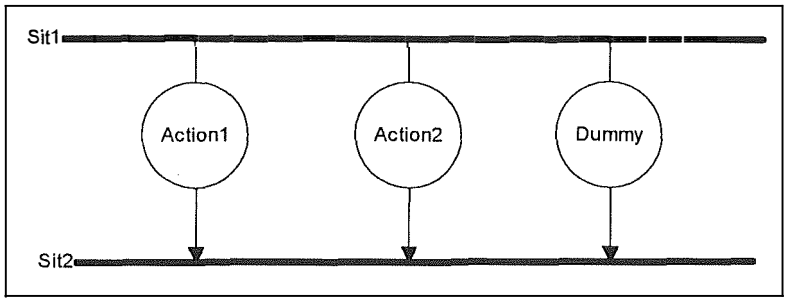


Figure 3-12 : Dummy Action.

3.2.9.3.2 « *Action Composition* » composée d'actions externes et internes.

En ALBERT II, nous distinguons deux sortes d'actions externes : les actions déclencheurs (c'est-à-dire les actions dont la responsabilité du déclenchement appartient à l'agent exportateur), et les actions services(c'est-à-dire les actions dont le déclenchement est dû à l'agent importateur). Ces deux types d'actions se modélisent différemment en OBLOG. Dans les figures qui suivent, nous utilisons les notations *self*, *Objet*, *Objet1* et *Objet2* pour indiquer les appartenances des graphes de comportement. *Self* désigne l'objet que nous traitons.

3.2.9.3.2.1 Actions déclencheurs.

Nous traduirons ce type d'actions par un appel vers l'action de l'objet qui suit l'action déclencheur. Nous vous présentons des exemples à la Figure 3-13 pour la conjonction et à la Figure 3-14 pour la séquence.

La conjonction.

En ALBERT II :

```
Action ↔ Agent1.Action1 ⊗ Action2
```

En OBLOG :

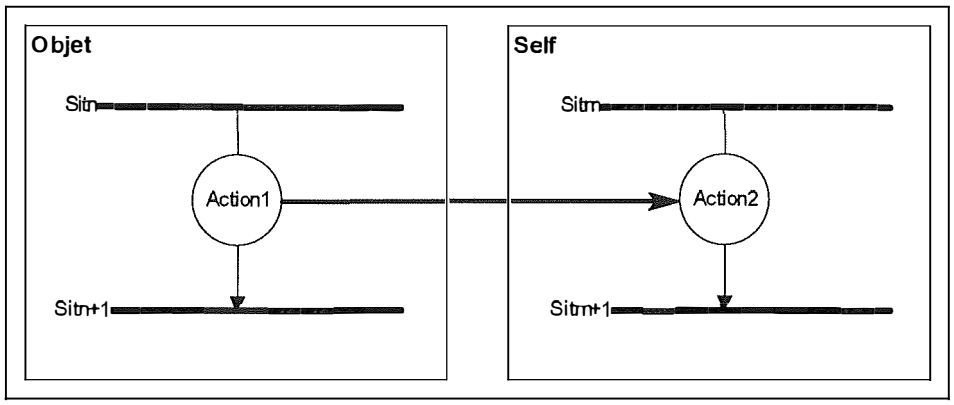


Figure 3-13 : Conjonction d'actions internes et externes.

Il est à noter que *Action2* hérite des « *Effect of Action* » de *Action1* dans *Self*.

La séquence.

En ALBERT II :

```
Action ↔ Agent1.Action1 ; Action2
```

En OBLOG :

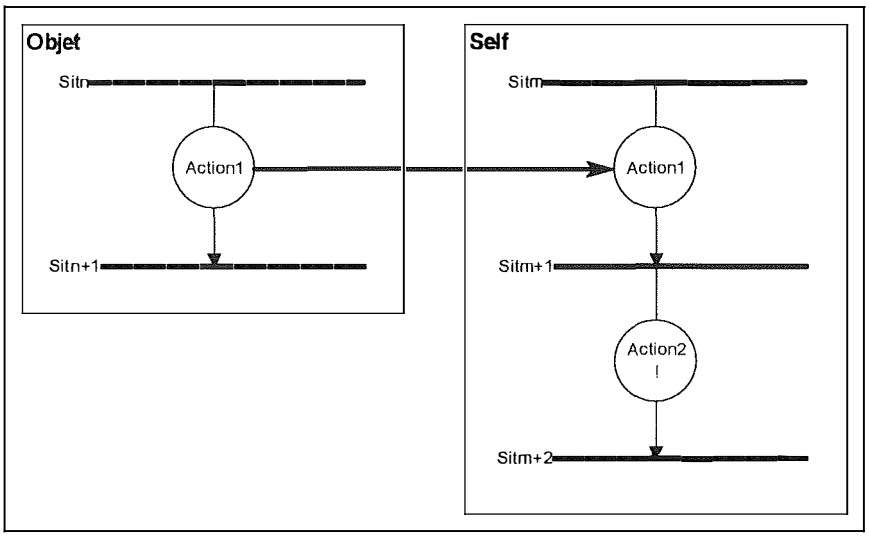


Figure 3-14 : Séquence d'actions internes et externes.

3.2.9.3.2.2 Actions services.

La traduction de ce type de compositions diffère des actions déclencheurs uniquement par l'objet qui déclenche l'action. Dans ce cas-ci, c'est l'agent importateur qui sera responsable de l'occurrence de l'action. Donc, c'est son correspondant OBLOG qui réalisera l'appel. Nous vous présentons des exemples de conjonction et de séquence à la Figure 3-15 et à la Figure 3-16.

La conjonction.

En ALBERT II :

```
Action ↔ Action1 ⊗ Agent.Action2
```

En OBLOG :

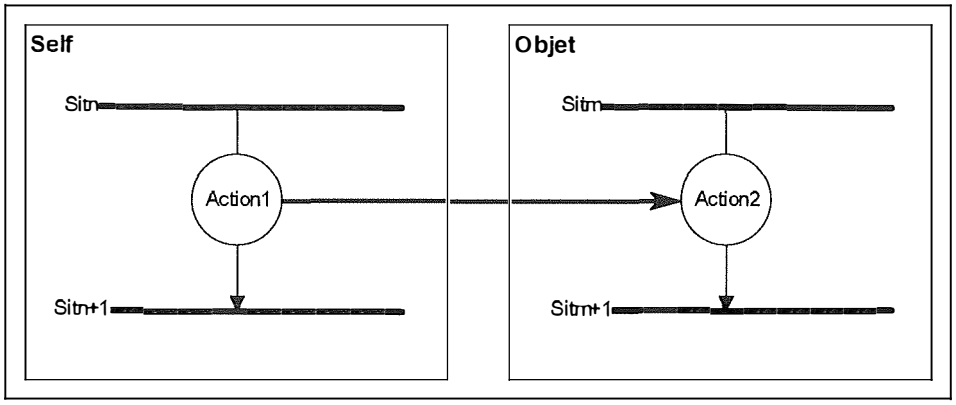


Figure 3-15 : Conjonction d'actions internes et externes.

Dans ce cas-ci également, *Action2* hérite des « *Effect of Action* » de *Action1*.

La séquence.

En ALBERT II :

```
Action ↔ Action1 ; Agent.Action2
```

En OBLOG :

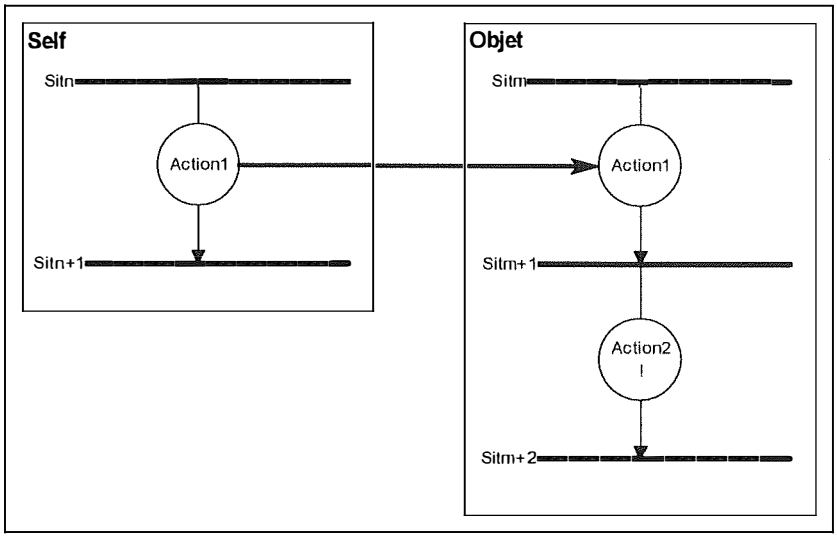


Figure 3-16 : Séquence d'actions internes et externes.

3.2.9.3.3 « *Action Composition* » composée d'actions externes.

Dans ce cas-ci également, nous ne traiterons que les cas de la séquence et de la conjonction.

La conjonction.

En ALBERT II :

```
Action ↔ Agent1.Action1 ⊗ Agent2.Action3
```

En OBLOG :

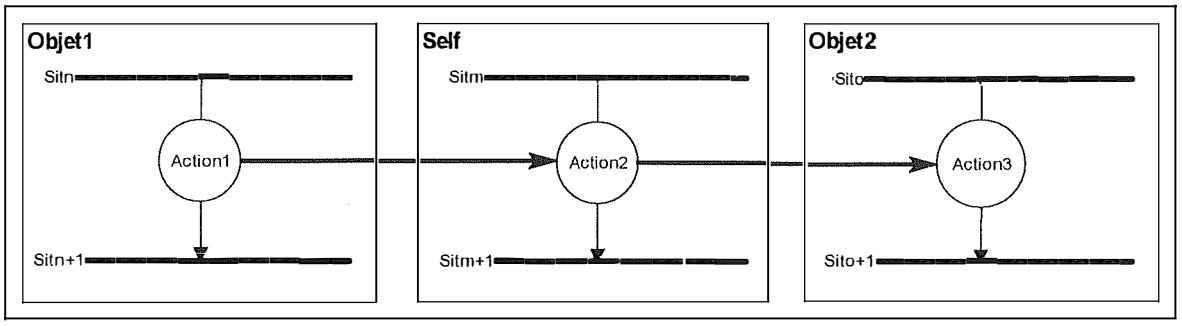


Figure 3-17 : Conjonction d'actions externes.

Action2 hérite à la fois des « *Effect of Action* » de *Action1* et de *Action2*.

La séquence.

En ALBERT II :

```
Action ↔ Agent1.Action1 ; Agent2.Action2
```

En OBLOG :

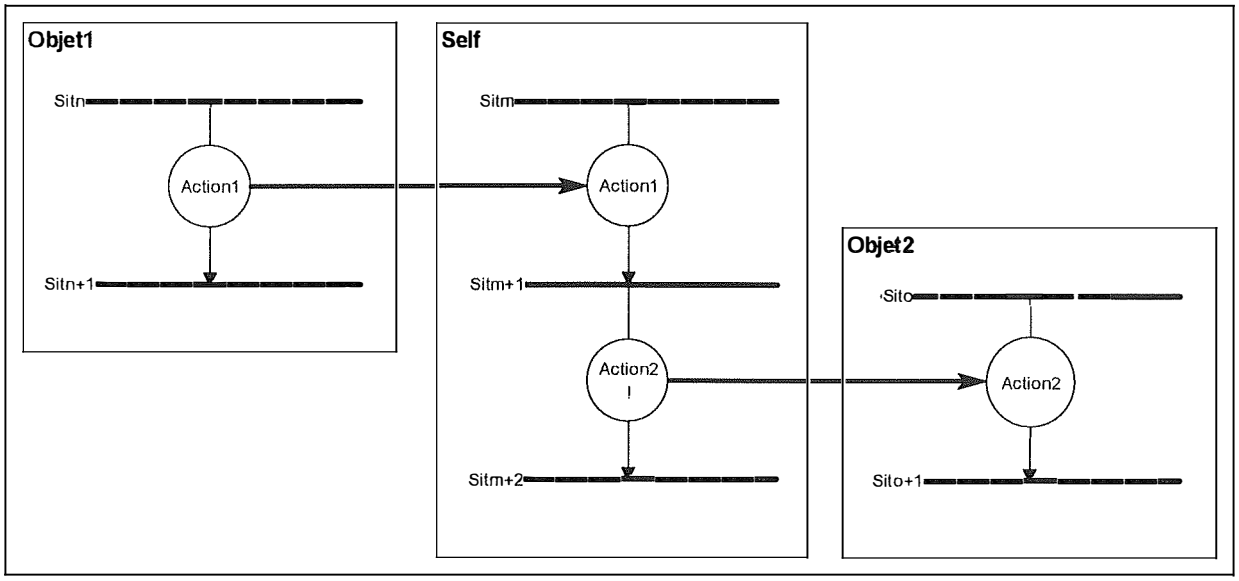


Figure 3-18 : Séquence d'actions externes.

3.2.10 INTEGRATION LOCALE.

Le but de cette phase est de créer tous les sous-graphes complets c'est-à-dire que certaines traductions locales couvrent les mêmes transitions. Il faut alors les intégrer dans un même sous-graphe. Un exemple vous est présenté à la section 3.3.8.

3.2.11 INTEGRATION GLOBALE.

Dans cette section, nous présentons comment intégrer les résultats de la traduction des contraintes en graphe comportemental au niveau global.

Pour commencer, il faut créer une situation de naissance ainsi qu'une transition de naissance. Ensuite, sur la première situation, nous pouvons ajouter tous les sous-graphes réalisés lors de la phase de traduction locale. Nous refermons ces sous-graphes si ce n'est déjà fait (c'est-à-dire que la dernière transition du sous-graphe aura comme destination la situation origine de ce sous-graphe). Une fois ce premier travail réalisé, il faut ajouter les transitions pouvant se réaliser à n'importe quel moment à chaque situation. De plus, il faudra intégrer les sous-graphes non encore présents dans chaque sous-graphe. Et ainsi de suite, jusqu'à ce que

chaque sous-graphe contienne tous les autres sous-graphes produits lors de la phase de traduction.

Malheureusement, cette méthode ne produit qu'un sous-ensemble des *interleaving* possibles. Il est dès lors nécessaire de compléter chaque sous-graphe par des transitions pour couvrir l'entièreté des *interleaving*.

3.2.12 CREATION DES APPELS.

Cette étape consiste à créer tous les appels entre les différents objets. Nous avons déjà présenté en brefles appels que l'objet OBSERVATEUR devra réaliser. Il faudra en plus pour tous les objets créer des appels pour les mises à jour d'attributs d'autres objets et pour les déclenchements d'actions. Une partie de ces appels est reprise de la traduction locale des contraintes comportementales.

3.2.13 TRADUCTION DES CONTRAINTES D'« ACTION DURATION ».

Comme nous l'avons dit, lorsqu'un prototype est développé, il y a un prix à payer. Dans notre cas, nous devons mettre de côté ce qui concerne les contraintes de temps. Donc, nous considérerons que toutes les actions de la spécification sont instantanées.

3.2.14 TRADUCTION DES CONTRAINTES DE COOPERATION.

Il existe quatre types de contraintes de coopération : *Action Information*, *Action Perception*, *State Information* et *State Perception*. Seules les *Action Information* sont directement modélisables en OBLOG. Cette traduction est réalisée au niveau de la condition d'appel. Pour ce qui est des autres types de contraintes, ceci est moins simple à réaliser.

L'*Action Perception* peut être traduite grâce à un attribut dans l'objet appelé de type booléen, qui indiquerait si l'objet est prêt à recevoir l'appel ou non. Mais le test sur cet attribut doit se faire dans l'objet appelant par le biais de la condition d'appel. En effet, si nous placions une condition sur les transitions associées à l'action appelée, la réalisation de l'action correspondante de l'objet appelant serait considérée comme n'ayant pas eu lieu puisque l'appel aurait échoué. Or, ce n'est pas le comportement que nous souhaitons reproduire.

Pour ce qui est des *State Information* et des *State Perception,* nous avons exploré plusieurs pistes mais elles ne se sont avérées valables qu'au cas par cas. Nous ne pouvons donc pas donner de marche à suivre pour modéliser ces contraintes.

3.2.15 TRADUCTION DES « DERIVED COMPONENTS ».

Les composants dérivés peuvent être remplacés en OBLOG par un attribut. Mais, il faudra soi-même le mettre à jour à chaque fois que nous désirerons consulter sa valeur.

3.3 EXEMPLE.

Prenons la spécification de l'ascenseur et appliquons lui cet algorithme en nous concentrant sur l'agent CABINE à partir de l'étape 2 puisque les seuls types construits sont des types énumérés et que leur transformation est triviale.

3.3.1 SUPPRESSION DES AGENTS EXTERNES.

Les agents UTILISATEURS et REPARATEUR sont des agents faisant partie de l'environnement du système. Pour l'agent UTILISATEURS , nous déplaçons ses actions vers d'autres agents tandis que pour l'agent REPARATEUR nous créons un interface.

L'agent CABINE reçoit les actions *Choisir_destination* et *Commander_ouverture*.

3.3.2 SUPPRESSION DE L'INDETERMINISME.

L'action *Detecter_panne* de l'agent CABINE introduit de l'indéterminisme puisqu'elle peut se produire n'importe quand. Pour supprimer cet indéterminisme, nous dédoublons cette action dans l'objet OBSERVATEUR.

3.3.3 IDENTIFICATION DES AGENTS.

L'agent CABINE sera du type DBX puisqu'il sert à héberger des actions de l'agent UTILISATEURS.

3.3.4 TRANSFORMATION DE L'AGENT EN OBJET.

Nous créons un objet nommé CABINE dont les actions seront *Detecter_panne*, *Reparer_cabine*, *Monter_cabine*, *Descendre_cabine*, *Choisir_destination*, *Commander_ouverture, Attendre_5s, Ouvrir_porte* et *Fermer_porte* ainsi que les attributs *Etat_porte*(BOOL), *Position*(ETAGES), *Panne_a_venir*(BOOL) et *Etages_arret* (LISTOFETAGES).

De plus, les actions devront mettre à jour les attributs comme dans la spécification ALBERT II.

```
Operation Detecter_panne
     Updates
     Panne_a_venir := TRUE
Operation Reparer_cabine
     Updates
     Panne_a_venir := FALSE
Operation Monter_cabine
     Updates
```

```
      Position := ONE[Etages | Numero = self.Position.Numero + 1]
Operation Descendre_cabine
      Updates
      Position := ONE[Etages | Numero = self.Position.Numero - 1]
Operation Choisir_destination
      Parameters
      Etage      :     Etages
      Updates
      Etages_arret := Append(Etages_arret,
Choisir_destination.Etage)
Operation Ouvrir_porte
      Updates
      Etat_porte := TRUE
      Etages_arret := Remove(Etages_arret, Position)
Operation Fermer_porte
      Updates
      Etat_porte := FALSE
```

Nous remarquons que les opérations *Prec* et *Suiv* ont été remplacées par des requêtes OBLOG.

3.3.5 AJOUT DES ACTIONS DE NAISSANCE ET DE MORT.

Nous ajoutons une action *Naitre* et une action *Mourir*.

```
Operation Naitre
      Updates
      Etat_porte := FALSE
      Panne_a_venir := FALSE
```

3.3.6 CONSTRUCTION DE L'OBJET OBSERVATEUR.

Nous ne présentons pas le résultat de cette étape. Mais il est possible de consulter le résultat final en annexe.

3.3.7 TRADUCTION LOCALE.

```
STATE BEHAVIOUR

Panne_a_venir ⇒ ◊ ⌉ Panne_a_venir
```

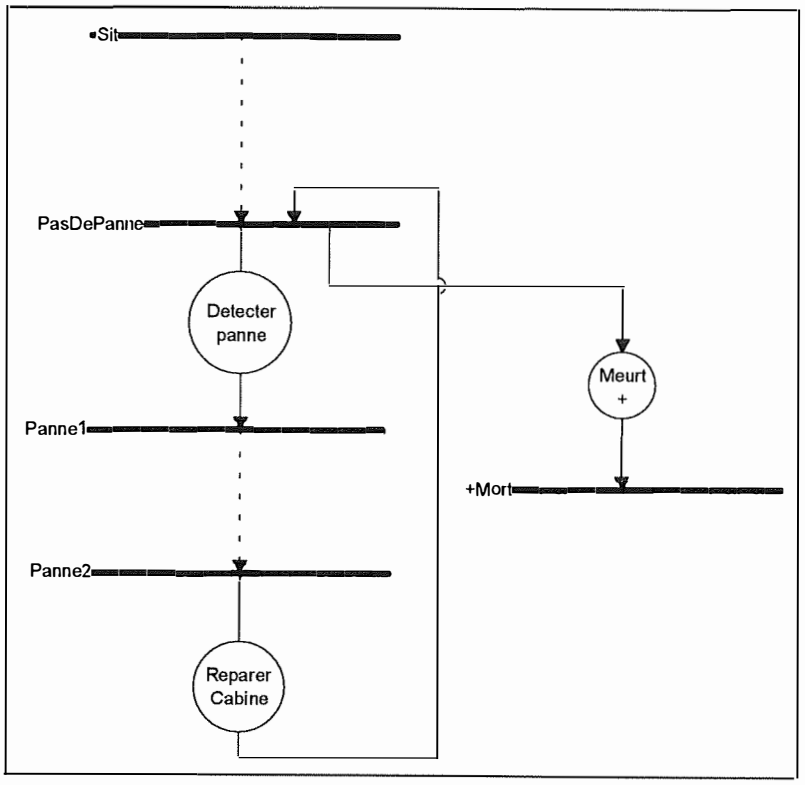


Figure 3-19 : Traduction du State Behaviour.

$$Etat\_porte \Rightarrow \Diamond \lceil Etat\_porte$$

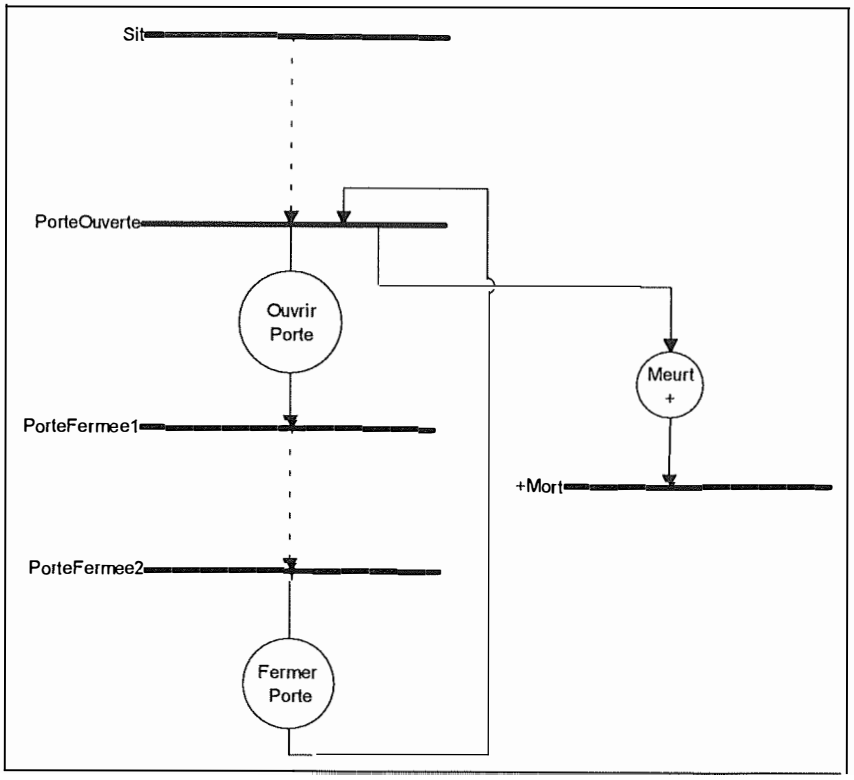


Figure 3-20 : Traduction du State Behaviour.

ACTION COMPOSITION

```
Ouvrir ↔ Utilisateurs.Commander_ouverture ; Contrôle.Ouvrir_porte
Prolonger ↔ Utilisateurs.Commander_ouverture ;
Contrôle.Attendre_5s
```

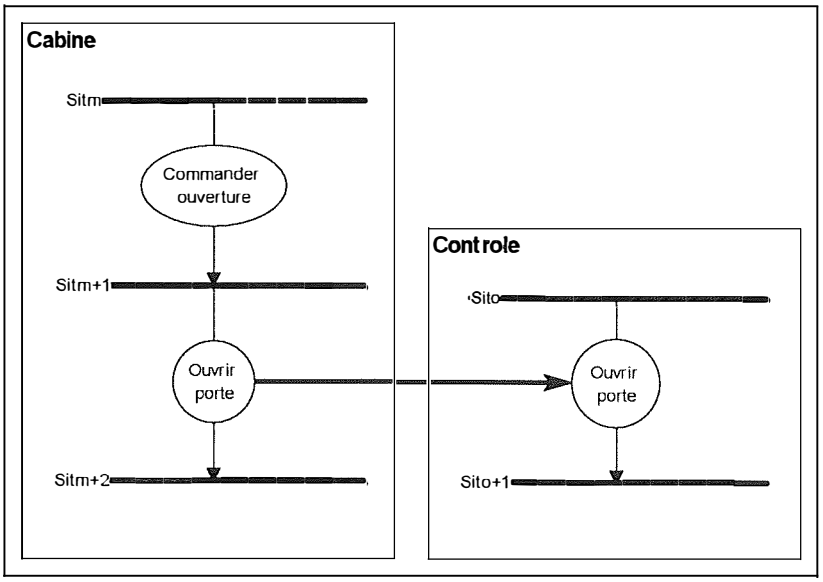


Figure 3-21 : Traduction de l'Action Composition.

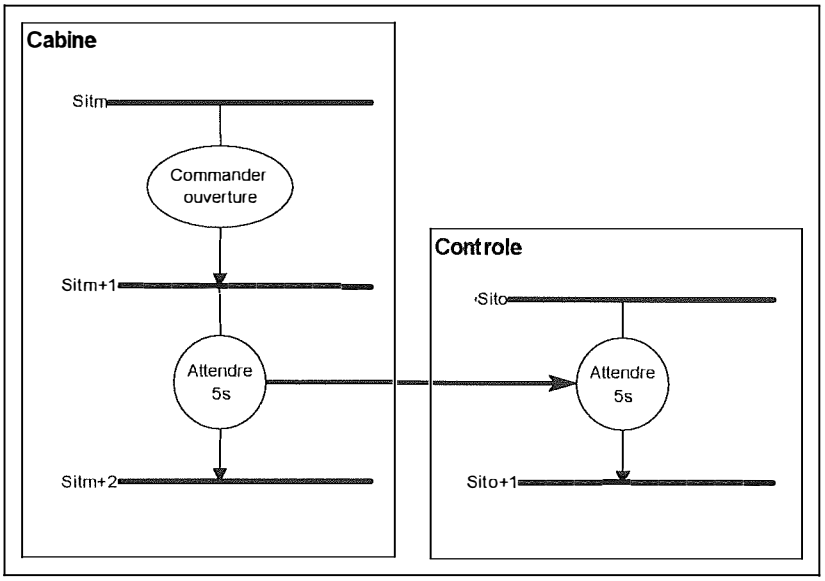
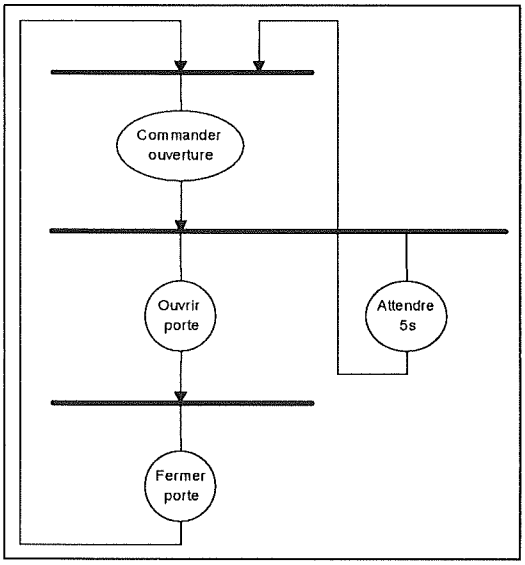
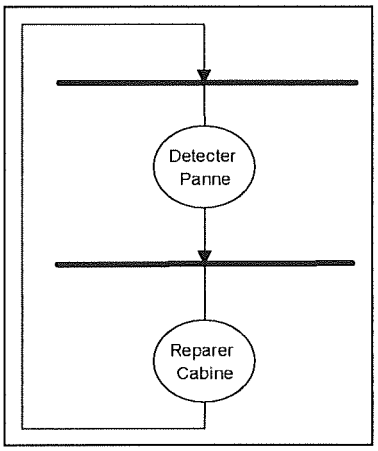


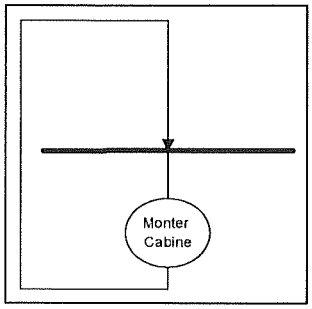
Figure 3-22 : Traduction de l'Action Composition.

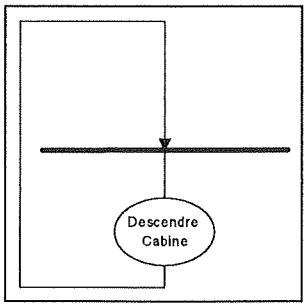
3.3.8 INTEGRATION LOCALE.

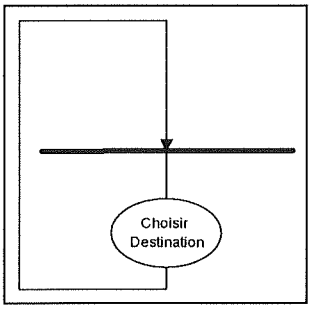
Nous vous présentons ici le résultat de l'intégration locale. Il s'agit de 5 sous-graphes que nous devons encore intégrer ensemble.





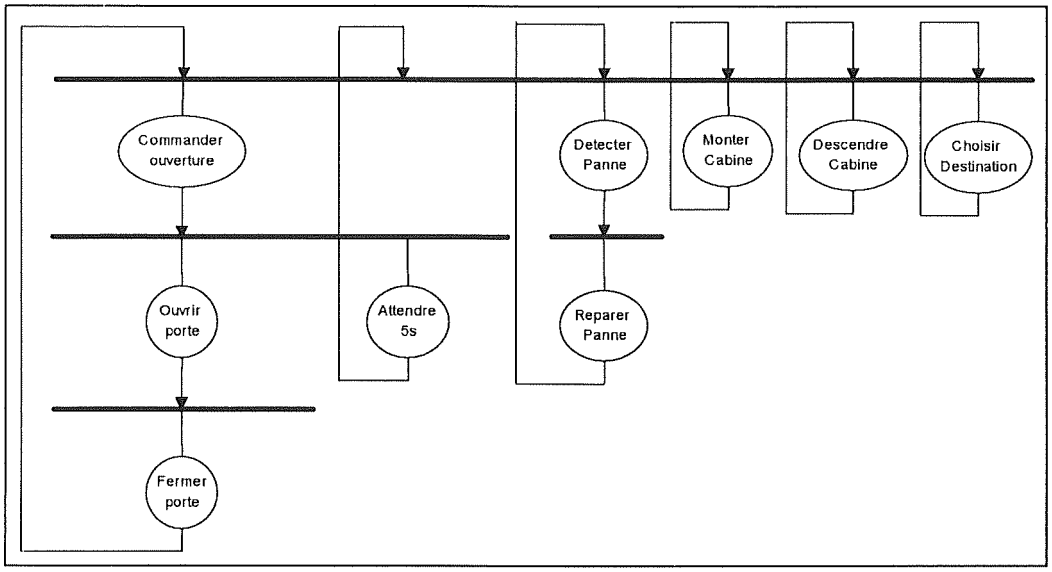






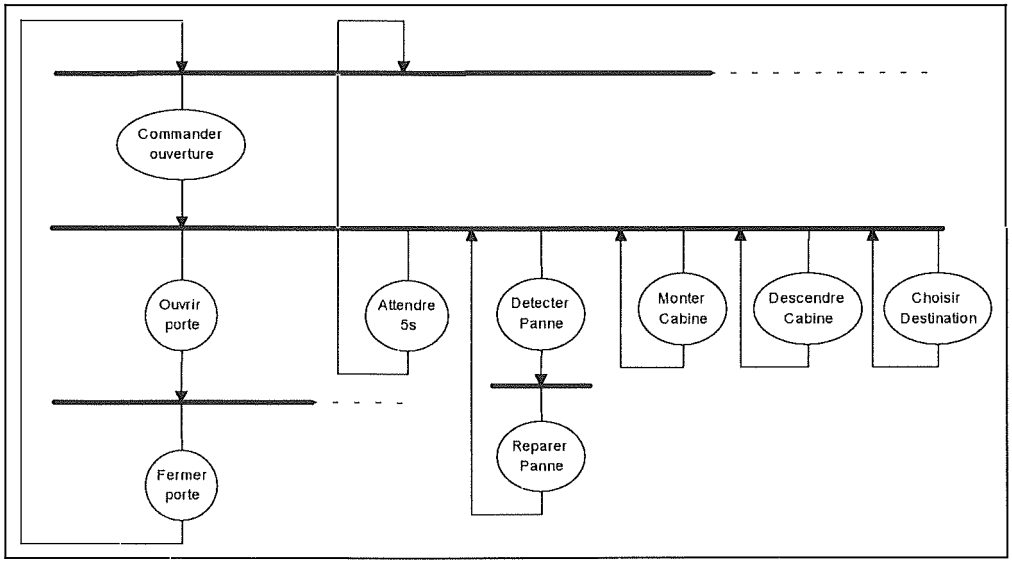
3.3.9 INTEGRATION GLOBALE.

Voici le résultat de la première phase de l'intégration globale :

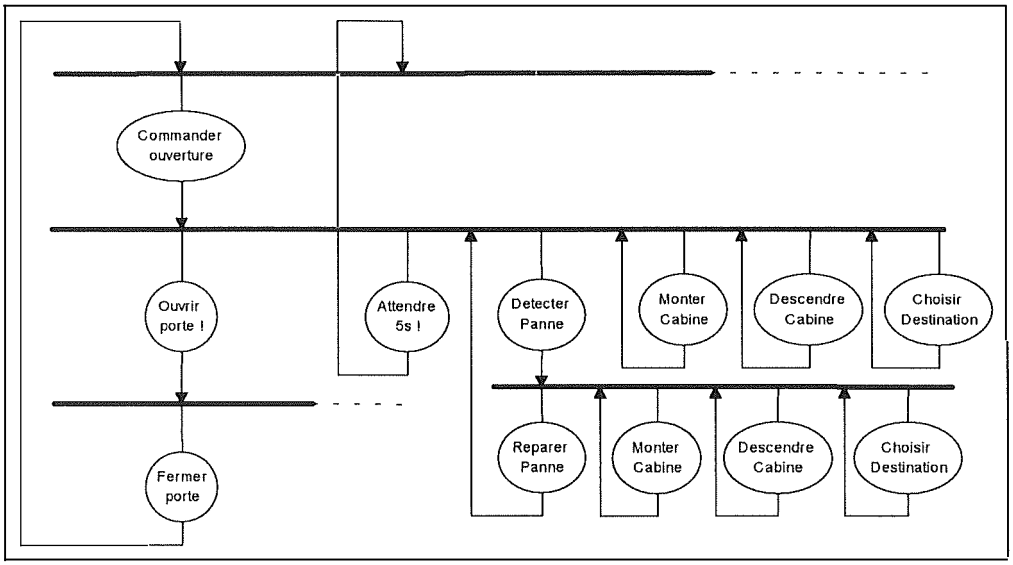


Il faut cependant encore ajouter deux situations et trois transitions pour les actions de naissance et de mort (2 transitions). La destination de la transition de naissance est la première situation de ce graphe et les origines des transitions de mort sont cette même première situation et la situation origine de *Attendre_5s*.

Il faut maintenant compléter les sous-graphes en ajoutant à chaque situation les sous-graphes de la phase précédente.



Il faut recommencer ce qui a été fait à l'étape précédente jusqu'à ce qu'il n'y ait plus de sous-graphe ne comportant pas tous les autres sous-graphes.



Il faudrait en plus de ceci permettre tous les interleaving possibles des séquences *Commander_ouverture; Ouvrir_porte ;Fermer_porte* et *Detecter_panne ; Repare_panne*. Ceci peut être réalisé simplement en ajoutant des transitions.

3.3.10 CREATION DES APPELS.

L'action *Ouvrir_porte* et *Attendre_5s* sont des actions déclencheurs, cela donne donc lieu à des appels. Ces appels auront comme destinataire CONTROLE et ces actions seront des actions actives.

CONCLUSION.

APPORTS ET PERSPECTIVES DE CE TRAVAIL.

Ce mémoire fournit une méthode pour développer des prototypes en OBLOG de spécifications de cahiers des charges en ALBERT II. Ceci apporte à l'informaticien un outil d'aide à la validation, dans un premier temps par rapport au modèle mental qu'il s'était fait du système, dans un second temps par rapport aux besoins effectifs des clients. Il serait toute fois intéressant de développer une preuve du prototype (c'est-à-dire de s'assurer qu'il y a bien correspondance sémantique entre le prototype et la spécification).

Ce travail a également permis de commenter les langages ALBERT II et OBLOG notamment à travers l'étude de cas réalisée. Cette étude a mis en évidence certaines lacunes tant pour ALBERT II que pour OBLOG. Rappelons par exemple, pour ALBERT II, l'absence de contraintes globales et de pseudo-polymorphisme. Pour OBLOG, nous pouvons citer le peu de constructeurs de types, l'absence d'opérations définissables sur les types construits, la présence d'un trop grand nombre d'artefacts ou encore la discontinuité entre le *Community Diagram* et le *Declaration Diagram*. Ces commentaires pourraient servir de bases aux prochaines évolutions de ces langages.

Il est à noter que certaines phases de cette méthodologie n'ont été développées que succinctement, il serait donc intéressant de les approfondir. Il s'agit entre autres de la phase d'intégration globale et de la phase de traduction des contraintes de coopération portant sur le *State information* et le *State Perception*. Il semblerait que derrière la phase d'intégration globale, le concept d'*interleaving* joue un rôle important.

BIBLIOGRAPHIE.

OUVRAGES CONCERNANT ALBERT.

[DUE94] Eric Dubois, Philippe Du Bois and Fréderic Dubru, « Animating Formal Requirements Specifications of Co-operative Information Systems », in Proceedings of the Second International Conference on Co-operative Information Systems - CoopIS-94, pages 101-112, Toronto (Canada), May 17-20, 1994. University of Toronto Press inc.

[DUB95] Philippe Du Bois, « The ALBERT II Language », Course, Computer science Department, University of Namur, Namur (Belgique) May 1995.

[JUN96] Bernard Jungen, « The ALBERT II Reference Manual », version 1.1, January-February 1996.

[DUB94] Philippe Du Bois, « The ALBERT II Language : On the Design and the Use of a Formal Specification Language for Requirements Analysis », PhD Thesis, Computer Science Department, University of Namur, Namur (Belgique), September 1994.

[ICA95] E. Dubois, J. Hagelstein, A. van Lamsweerde, F. Orejas, J. Souquieres, P. Wodon, « A Guided Tour through the ICARUS Project », Software Engineering Notes, vol. 20, no. 2, pp. 28-33, 1995.

[DDD94] Eric Dubois, Philippe Du Bois, Frederic Dubru, « Animating Formal Requirements Specifications of Co-operative Information Systems », Computer Science Department, University of Namur, Namur (Belgique), May 1994.

OUVRAGES CONCERNANT OBLOG.

[OBL95-1] « OBLOG CASE V1.2, Introduction », OBLOG Software, July 1995.

[OBL95-2] « OBLOG CASE V1.2, Reference Manuel », OBLOG Software, June 1995.

[OBL95-3] « OBLOG CASE V1.2, Relaese Notes », OBLOG Software, June 1995.

[OBL95-4] « OBLOG CASE V1.2, User's Guide », OBLOG Software, June 1995.

[OBL95-5] « OBLOG CASE V1.2, The OBLOG Method : An Overview », OBLOG Software, June 1995.

[OBL95-6] « OBLOG, Language Reference », OBLOG Software, June 1995.

[SER94-1] Cristina Sernadas, Paula Gouveia, Amílcar Sernadas, « OBLOG : Object-oriented, Logic-based Conceptual Modeling », Departemento de Matemática, Instituto Superior Técnico, Lisboa (Portugal), November 1994.

[SER94-2] Almílcar Sernadas, J.F. Costa, Cristina Sernadas, « Object Specification through Diagrams - OBLOG Approach », Technical Report, OBLOG Software, Lisboa (Portugal), 1994.

[ZEI96] Jean-marc Zeippen, « Introduction au langage de spécification OBLOG », Course, Computer Science Department, University of Namur, Namur (Belgique), February 1996.

OUVRAGES CONCERNANT ALBERT & OBLOG.

[DUB93] Frédéric Dubru, « Prototypage de Spécifications Formelles des Besoins », Mémoire présenté en vue de l'obtention du diplôme de Maître en Informatique, Institut d'Informatique, Facultés Universitaires N. D. de la Paix, Namur (Belgique), Septembre 1993.

[DDE94] Michel Degroot & Gilles Delcourt, « Conception et Implémentation d'une Application Distribuée dans le Domaine de la Productique», Mémoire présenté en vue de l'obtention du diplôme de Maître en Informatique, Institut d'Informatique, Facultés Universitaires N. D. de la Paix, Namur (Belgique), Septembre 1994.

OUVRAGES CONCERNANT L'INGENIERIE DU LOGICIEL.

[BOE81] Barry W. Boehm, « Software Engineering Economics », Prentice Hall, 1981.

[ROY70] W.W.Royce, « Managing the Development of Large Software Systems : Concepts and Techniques", Proceedings, WESCON, August 1970.

[DBB97] Bénédicte Dano, Henri Briand, Franck Barbier, « Producing object-oriented dynamic specification : an approach based on the concept of *use case* », to appear in RE'97, Washington (USA), January 1997.

OUVRAGES CONCERNANT L'ANALYSE ET LA CONCEPTION ORIENTEE-OBJET.

[HIL96] David R.C. Hill, « Object-Oriented Analysis and Simulation », Addison-Wesley, 1996.

[RUM91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen, « Object-Oriented Modeling and Design », Prentice-Hall, 1991.

OUVRAGE CONCERNANT LE PROTOTYPAGE.

[HAB90] Naji Habra, « A Transformational Method for Functional Prototyping », PhD Thesis, Computer Science Department, University of Namur, Namur (Belgique), September 1990.

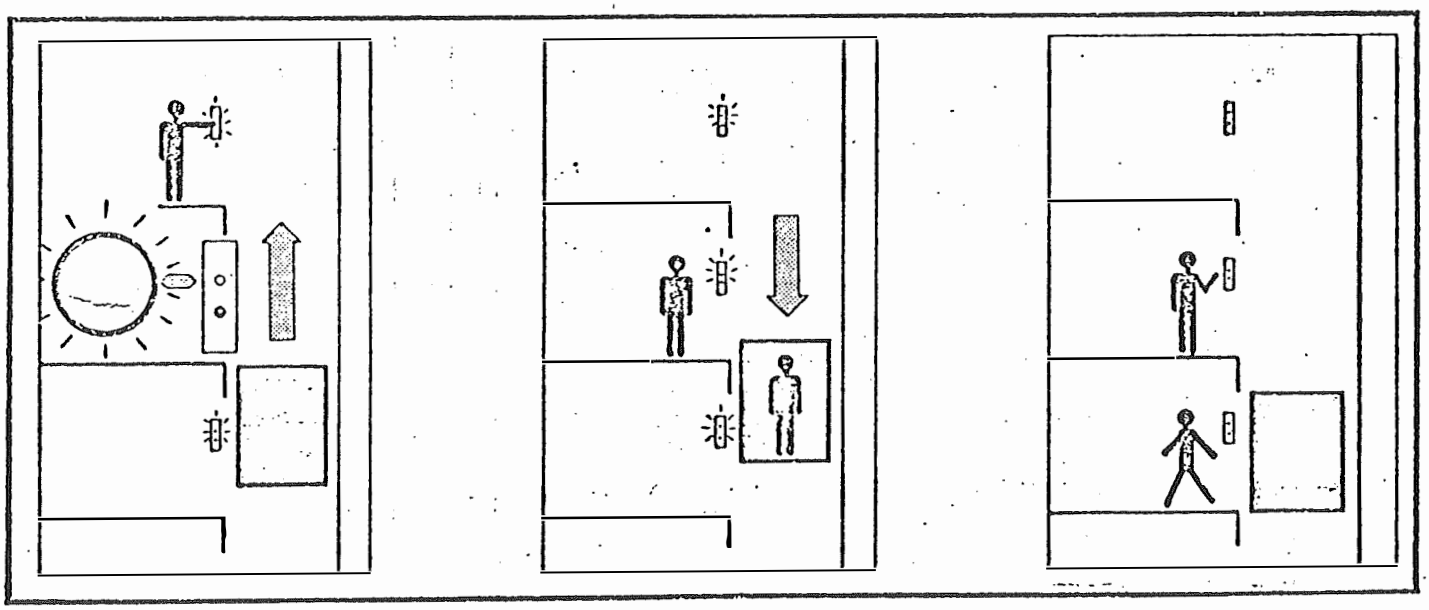
ANNEXES.

DOCUMENTATION SUR LE COMPORTEMENT DES ASCENSEURS.

MANOEUVRE UNIVERSELLE A BLOCAGE

Le passager provoque la mise en route de la cabine par appui sur un bouton d'envoi en cabine ou un bouton d'appel palier. La cabine s'arrête d'elle-même à l'étage correspondant à la commande réalisée. Toute autre commande est rendue inopérante tant que la manoeuvre n'est pas complètement exécutée.

Il est prévu un dispositif de retardement, destiné à empêcher toute manoeuvre, à partir des boutons paliers pendant quelques secondes après l'arrêt de la cabine à un étage, afin de laisser le temps à une personne de pénétrer dans la cabine et d'y effectuer sa commande.



Appel à un étage.
Illumination du signal d'occupation.

Ascenseur occupé.
Ne peut être appelé à un autre niveau.
Occupant seul maître à bord.

Ascenseur libre.
Signaux lumineux éteints.
L'ascenseur peut être appelé.

OTIS

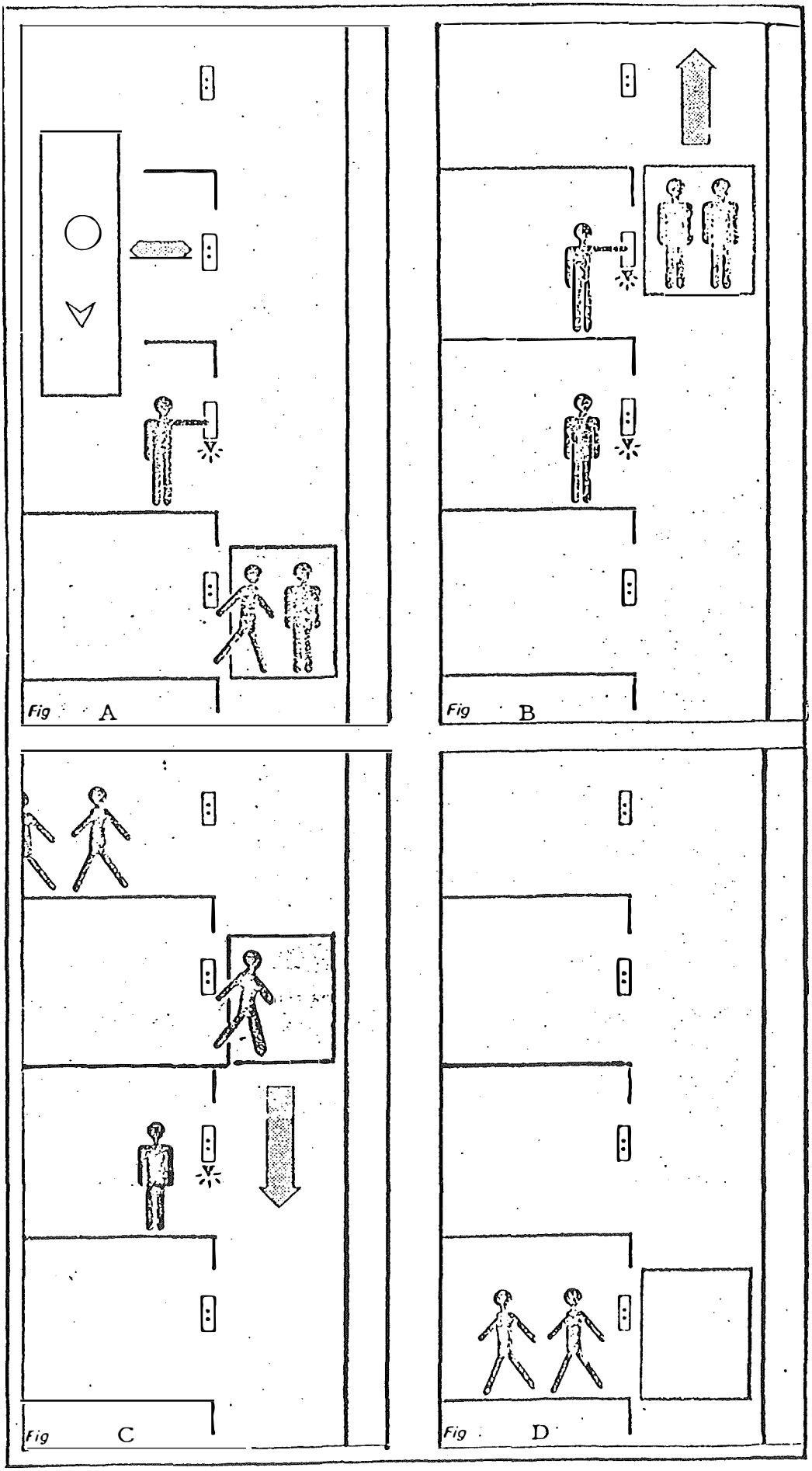
MANOEUVRE COLLECTIVE DESCENTE

A. Les passagers entrent dans la cabine et appuyent les boutons de leurs étages de destination.

B. Pendant le trajet en montée, les appels paliers pour monter sont collectés dans la cabine.

C. Pendant la course descente, tous les appels paliers pour descendre sont collectés par la cabine.

D. Les passagers quittent la cabine à leur étage de destination.



OTIS

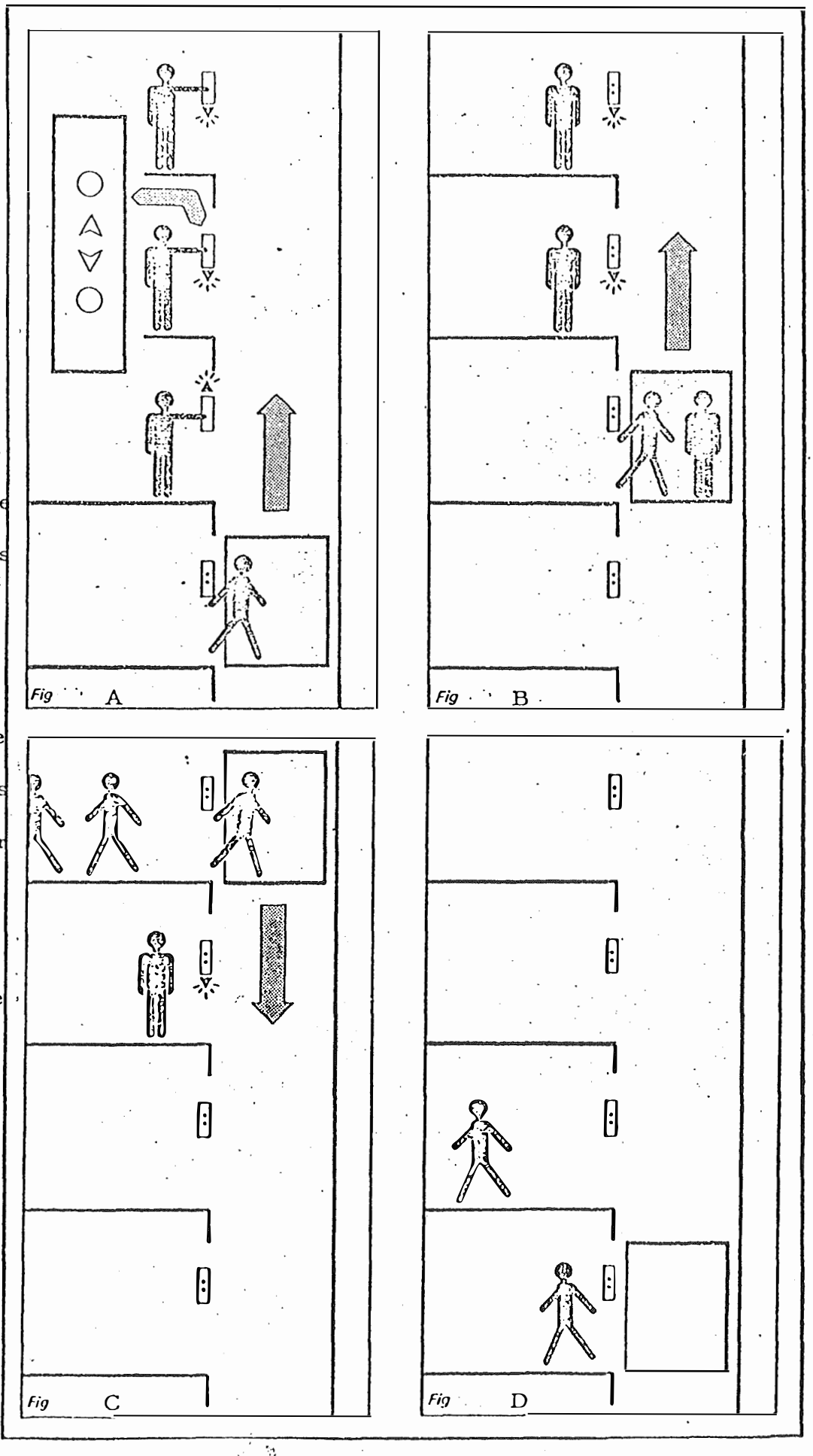
MANOEUVRE COLLECTIVE MONTEE DESCENTE

A. Les passagers entrent dans la cabine et appuyent les boutons de leurs étages de destination.

B. Pendant la course en montée, tous les appels paliers pour monter sont collectés dans la cabine.

C. Pendant la course descente, tous les appels paliers pour descendre sont collectés par la cabine.

D. Les passagers quittent la cabine à leur étage de destination.



OTIS

SPECIFICATION DU SYSTEME EN ALBERT II.

SPECIFICATION FAC

CONSTRUCTED TYPES

```
TETATMOTEUR=Enum[ARRETE,MONTANT,DESCENDANT]
TPHASE=Enum[STABLE,PREPAREUP,PREPAREDOWN,UP,DOWN]
TSENS=Enum[MONTEE,DESCENTE]
```

OPERATIONS

- PremierEtage : → ETAGE
  PremierEtage = val
  with val <> DernierEtage

- DernierEtage : → ETAGES

- Avant : ETAGES × ETAGES → BOOLEAN
  Avant(a , b) = val
  with [val = FALSE ⇐ b = PremierEtage]
  ∧ [val = TRUE ⇔ a = Prec(b) ∨ Avant(a,Prec(b))]

- Prec : ETAGES → ETAGES*
  Prec (a) = val
  with (val = UNDEF ⇔ a = PremierEtage)
  ∧ (¬∃ b : a <> b ∧ Prec(a)=Prec(b))

- Suiv : ETAGES → ETAGES*
  Suiv (a) = val
  with (val = UNDEF ⇔ a = DernierEtage)
  ∧ (¬∃ b : a <> b ∧ Suiv(a)=Suiv(b))
  ∧ (x <> PremierEtage ⇒ x = Suiv(Prec(x)))

SOCIETY FAC

```
(CONTROLE)
(MOTEUR)
(UTILISATEURS)))
(CABINE)
(REPARATEUR)
(ETAGES)))
```

AGENT FAC/CONTROLE

STATE COMPONENTS

```
Phase instance of TPHASE                    → FAC/ETAGES
Panne_a_venir instance of BOOLEAN           → FAC/REPARATEUR
Prevenu instance of BOOLEAN
```

ACTIONS

```
Arret
*Attendre
*Attendre_5s                                → FAC/CABINE*
Changer_Phase(TPHASE)
*Commander_arret                            → FAC/MOTEUR
Commander_descente                          → FAC/MOTEUR*
Commander_montee                            → FAC/MOTEUR*
Detecter_panne
*Fermer_porte                               → FAC/CABINE,FAC/ETAGES
Ouvrir
*Ouvrir_apres_arret
*Ouvrir_porte                               → FAC/CABINE*,FAC/ETAGES*
Prevenir                                    → FAC/REPARATEUR*
Mise_a_jour
```

BASIC CONSTRAINTS

INITIAL VALUATION

```
Phase = STABLE
Panne_a_venir = FALSE
Prevenu = FALSE
```

LOCAL CONSTRAINTS

STATE BEHAVIOUR

```
Panne_a_venir ⇒ ◊ ¬Panne_a_venir
```

EFFECTS OF ACTIONS

```
Detecter_panne : Panne_a_venir := TRUE
Reparateur.Reparer_controle : Panne_a_venir := FALSE
Changer_Phase( D ) : Phase := D
Prevenir : Prevenu := TRUE
Mise_a_jour : Prevenu := FALSE
```

CAPABILITY

- XO(Changer_phase(UP) |
  (Phase = STABLE ∧ (∃ Etage : Avant(Cabine.Position,Etage) ∧
  (Etage.Appel_montee ∨ Cabine.Etages_arret[Etage])))
  ∨
  (Phase = PREPAREUP ∧ (∀ Etage : Avant(Etage,Cabine.Position) ⇒
  ¬Etage.Appel_montee))
  ∨
  (Phase = DOWN ∧ (∀ Etage : Avant(Etage,Cabine.Position) ⇒
  (¬Etage.Appel_descente ∧ ¬Etage.Appel_montee ∧
  ¬Cabine.Etages_arret[Etage])) ∧ (∃ Etage :
  Avant(Cabine.Position,Etage) ∧ Etage.Appel_montee) ∧
  ¬Cabine.Position.Appel_descente ∧
  ¬Cabine.Etages_arret[Cabine.Position]))
- XO(Changer_phase(DOWN) |
  (Phase = STABLE ∧ (∃ Etage : Avant(Etage,Cabine.Position) ∧
  (Etage.Appel_descente ∨ Cabine.Etages_arret[Etage])))
  ∨
  (Phase = PREPAREDOWN ∧ (∀ Etage : Avant(Cabine.Position,Etage)
  ⇒ ¬Etage.Appel_descente))
  ∨
  (Phase = UP ∧ (∀ Etage : Avant(Cabine.Position,Etage) ⇒
  (¬Etage.Appel_descente ∧ ¬Etage.Appel_montee ∧
  ¬Cabine.Etages_arret[Etage])) ∧ (∃ Etage :
  Avant(Etage,Cabine.Position) ∧ Etage.Appel_descente) ∧
  ¬Cabine.Position.Appel_montee ∧
  ¬Cabine.Etages_arret[Cabine.Position]))
- XO(Changer_phase(PREPAREDOWN) |
  (Phase = STABLE ∧ (∃ Etage : Avant(Cabine.Position,Etage) ∧
  Etage.Appel_descente))
  ∨
  (Phase = UP ∧ (∃ Etage : Avant(Cabine.Position,Etage) ∧
  Etage.Appel_descente) ∧ (∀ Etage : Avant(Cabine.Position,Etage)
  ⇒ (¬Etage.Appel_montee ∧ ¬Cabine.Etages_arret[Etage])) ∧
  ¬Cabine.Position.Appel_descente ∧
  ¬Cabine.Etages_arret[Cabine.Position]))
- XO(Changer_phase(PREPAREUP) |
  (Phase = STABLE ∧ (∃ Etage : Avant(Etage,Cabine.Position) ∧
  Etage.Appel_montee))
  ∨
  (Phase = DOWN ∧ (∃ Etage : Avant(Etage,Cabine.Position) ∧
  Etage.Appel_montee) ∧ (∀ Etage : Avant(Etage,Cabine.Position) ⇒
  (¬Etage.Appel_descente ∧ ¬Cabine.Etages_arret[Etage])) ∧
  ¬Cabine.Position.Appel_montee ∧
  ¬Cabine.Etages_arret[Cabine.Position]))
- XO(Changer_phase(STABLE) | ((Phase = UP ∨ Phase = DOWN) ∧ (∀
  Etage : ¬Etage.Appel_montee ∧ ¬Etage.Appel_descente ∧
  ¬Cabine.Etages_arret[Etage]))

∨ (Phase = UP ∧ (∃ Etage : Avant(Etage,Cabine.Position) ∧ Etage.Appel_montee) ∧ (∀ Etage: ¬Etage.Appel_descente ∧ ¬Cabine.Etages_arret[Etage]) ∧ (∀ Etage : Avant(Cabine.Position,Etage) ⇒ ¬Etage.Appel_montee) ∧ ¬Cabine.Position.Appel_montee)
∨ (Phase = DOWN ∧ (∃ Etage : Avant(Cabine.Position,Etage) ∧ Etage.Appel_descente) ∧ (∀ Etage: ¬Etage.Appel_montee ∧ ¬Cabine.Etages_arret[Etage]) ∧ (∀ Etage: Avant(Etage,Cabine.Position) ⇒ ¬Etage.Appel_descente) ∧ ¬Cabine.Position.Appel_descente))

- XO(Commander_arret | (Phase = UP ∧ Cabine.Position.Appel_montee) ∨ (Phase = DOWN ∧ Cabine.Position.Appel_descente) ∨ Cabine.Etages_arret[Cabine.Position])
- XO(Commander_descente | ¬Cabine.Etat_porte ∧ ¬Cabine.Etages_arret[Cabine.Position] ∧ ((Phase = DOWN ∧ ¬Cabine.Position.Appel_descente) ∨ (Phase = PREPAREUP)))
- XO(Commander_montee | ¬Cabine.Etat_porte ∧ ¬Cabine.Etages_arret[Cabine.Position] ∧ ((Phase = UP ∧ ¬ Cabine.Position.Appel_montee) ∨ (Phase = PREPAREDOWN)))
- XO(Prevenir | ¬Prevenu ∧ (Panne_a_venir ∨ ∃ Etage : Etage.Panne_a_venir ∨ Moteur.Panne_a_venir ∨ Cabine.Panne_a_venir))
  {Cabine : CABINE, Moteur : MOTEUR, Etage : ETAGES}
- XO(Mise_a_jour | Prevenu ∧ ¬Panne_a_venir ∧ ∀ Etage : ¬Etage.Panne_a_venir ∧ ¬Moteur.Panne_a_venir ∧ ¬Cabine.Panne_a_venir)
  {Cabine : CABINE, Moteur : MOTEUR, Etage : ETAGES}
- XO(Ouvrir | Phase = STABLE ∧ (Cabine.Position.Appel_montee ∨ Cabine.Position.Appel_descente))
- F(Ouvrir_porte | Moteur.Etat <> ARRETE)
- F(Attendre_5s|¬Cabine.Etat_porte ∨ ¬Cabine.Position.Etat_porte)

ACTION COMPOSITION

Arret ↔ Commander_arret ; Ouvrir_apres_arret
Ouvrir_après_arret ↔ Ouvrir_porte ; Attendre_5s ; Attendre ; Fermer_porte
Ouvrir ↔ Ouvrir_porte ; Attendre_5s ; Attendre ; Fermer_porte
Attendre ↔ { Attendre_5s }^{n}

ACTION DURATION

| Commander_arret | = 0 s
| Commander_montee | = 0 s
| Commander_descente | = 0 s
| Ouvrir_porte | = 0 s
| Fermer_porte | = 0 s
| Attendre_5s | = 5 s
| Prevenir | = 0 s
| Detecter_panne | = 0 s
| Changer_phase(_) | = 0 s

Co-operation Constraints

Action Perception

```
XK(Reparateur.Reparer_controle | Panne_a_venir)
```

State Perception

```
K( Cabine.Etages_arret | TRUE)
K( Cabine.Position | TRUE )
K( Cabine.Etat_porte | TRUE)
K( x.Etat_porte | TRUE )
K( x.Appel_montee | TRUE)
K( x.Appel_descente | TRUE)
K( x.Panne_a_venir | TRUE )
```

Action Information

```
K( Prevenir.Reparateur | TRUE )
XK( Ouvrir_porte.Etage | Etage = Cabine.Position )
K( Ouvrir_porte.Cabine | TRUE)                    {Cabine : CABINE}
K( Fermer_porte.Etage | TRUE )
K( Commander_montee.Moteur | TRUE )
K( Commander_descente.Moteur | TRUE )
K( Commander_arret.Moteur | TRUE )
K( Attendre_5s | TRUE )
```

State Information

```
K( Panne_a_venir.Reparateur | TRUE )
K( Phase.Etage | TRUE )
```

AGENT FAC/MOTEUR

STATE COMPONENTS

Etat **instance of** TETATMOTEUR → FAC/CONTROLE, FAC/CABINE
Panne_a_venir **instance of** BOOLEAN → FAC/CONTROLE, FAC/REPARATEUR

ACTIONS

*Monter_cabine → FAC/CABINE
*Descendre_cabine → FAC/CABINE
Detecter_panne
Monter
Descendre

BASIC CONSTRAINTS

INITIAL VALUATION

Etat = ARRETE
Panne_a_venir = FALSE

LOCAL CONSTRAINTS

STATE BEHAVIOUR

Etat = MONTANT ⇒ ◊ Etat = ARRETE
Etat = DESCENDANT ⇒ ◊ Etat = ARRETE
Panne_a_venir ⇒ ◊ ¬Panne_a_venir

EFFECTS OF ACTIONS

Detecter_panne : Panne_a_venir := TRUE
Reparateur.Reparer_moteur : Panne_a_venir := FALSE
Controle.Commander_montee : Etat := MONTANT
Controle.Commander_descente : Etat := DESCENDANT
Controle.Commander_arret : Etat := ARRETE

ACTION COMPOSITION

Monter ↔ Controle.Commander_montee; Monter_cabine
Descendre ↔ Controle.Commander_descente; Descendre_cabine

Co-operation Constraints

Action Perception

```
XK( Controle.Commander_montee | Etat = ARRETE)
XK( Controle.Commander_descente | Etat = ARRETE)
XK( Controle.Commander_arret | Etat <> ARRETE )
XK( Reparateur.Reparer_moteur | Panne_a_venir)
```

Action Information

```
K( Monter_cabine.Cabine | TRUE )
K( Descendre_cabine.Cabine | TRUE )
```

State Information

```
K( Panne_a_venir.X | TRUE )
K( Etat.X | TRUE )
```

AGENT FAC/UTILISATEURS

ACTIONS

```
Appeler(TSENS, ETAGES)              → FAC/ETAGES
Choisir_destination(ETAGES)         → FAC/CABINE
Commander_ouverture                 → FAC/CABINE*
```

LOCAL CONSTRAINTS

ACTION DURATION

```
| Appeler( _ , _ ) | = 0 s
| Choisir_destination( _ ) | = 0 s
| Commander_ouverture | = 0 s
```

CO-OPERATION CONSTRAINTS

ACTION INFORMATION

```
XK( Appeler( _ , Etage1 ).Etage2 | Etage1 = Etage2 )
K( Choisir_destination( _ ).Cabine | TRUE )
K( Commander_ouverture.Cabine | TRUE )
```

AGENT FAC/CABINE

STATE COMPONENTS

```
Etat_porte instance of BOOLEAN     → FAC/CONTROLE
Position instance of ETAGES        → FAC/CONTROLE
Panne_a_venir instance of BOOLEAN → FAC/CONTROLE, FAC/REPARATEUR
Etages_arret table of BOOLEAN
             indexed by ETAGES     → FAC/CONTROLE
```

ACTIONS

```
Detecter_panne
Ouvrir
Prolonger
```

BASIC CONSTRAINTS

INITIAL VALUATION

```
Etat_porte = FALSE
Etages_arret[i] = FALSE
Panne_a_venir = FALSE
```

LOCAL CONSTRAINTS

STATE BEHAVIOUR

```
Panne_a_venir ⇒ ◊ ¬Panne_a_venir
Etat_porte ⇒ ◊ ¬Etat_porte
```

EFFECTS OF ACTIONS

```
Detecter_panne : Panne_a_venir := TRUE
Reparateur.Reparer_cabine : Panne_a_venir := FALSE
Moteur.Monter_cabine : Position := Suiv(Position)
Moteur.Descendre_cabine : Position := Prec(Position)
Utilisateurs.Choisir_destination( dest ) : Etages_arret[dest] :=
TRUE
Controle.Ouvrir_porte : Etat_porte := TRUE;
Etages_arret[position] := FALSE
Controle.Fermer_porte : Etat_porte := FALSE
```

ACTION COMPOSITION

```
Ouvrir ↔ Utilisateurs.Commander_ouverture ; Controle.Ouvrir_porte
Prolonger ↔ Utilisateurs.Commander_ouverture ;
Controle.Attendre_5s
```

Co-operation Constraints

Action Perception

XK(Reparateur.Reparer_Cabine | Panne_a_venir)
K(Utilisateur.Choisir_destination(dest) | ¬Etages_arret[dest])
XK(Utilisateur.Commander_ouverture | Moteur.Etat = ARRETE)
K(Moteur.Descendre_cabine | TRUE)
K(Moteur.Monter_cabine | TRUE)
XK(Controle.Fermer_porte | Etat_porte)
XK(Controle.Ouvrir_porte | ¬Etat_porte)
K(Controle.Attendre_5s | TRUE)

State Perception

K(Moteur.Etat = ARRETE | TRUE)

State Information

K(Panne_a_venir.X | TRUE)
K(Etages_arret.Controle | TRUE)
K(Position.x | TRUE)
K(Etat_porte.Controle | TRUE)

AGENT FAC/REPARATEUR

ACTIONS

*Reparer_controle	→ FAC/CONTROLE
*Reparer_moteur	→ FAC/MOTEUR
*Reparer_cabine	→ FAC/CABINE
*Reparer_etages	→ FAC/ETAGES
*Reparer1	
*Reparer2	
Reparer	

LOCAL CONSTRAINTS

CAPABILITY

F(Reparer_controle | ¬Controle.Panne_a_venir)
F(Reparer_moteur | ¬Moteur.Panne_a_venir)
F(Reparer_cabine | ¬Cabine.Panne_a_venir)
F(Reparer_etages | ∀ Etage : ¬Etages.Panne_a_venir)

ACTION COMPOSITION

Reparer ↔ Controle.Prevenir ; Reparer1
Reparer1 ↔ { Reparer2 } ^{n}
Reparer2 ↔
  Reparer_controle⊗Reparer_moteur⊗Reparer_cabine⊗Reparer_etages

CO-OPERATION CONSTRAINTS

ACTION PERCEPTION

K(Controle.Prevenir | TRUE)

STATE PERCEPTION

K(Controle.Panne_a_venir | TRUE)
K(Moteur.Panne_a_venir | TRUE)
K(Cabine.Panne_a_venir | TRUE)
K(Etages.Panne_a_venir | TRUE)

ACTION INFORMATION

K(Reparer_controle.Controle | TRUE)
K(Reparer_moteur.Moteur | TRUE)
K(Reparer_Cabine.Cabine | TRUE)
K(Reparer_Etages.Etage | TRUE)

AGENT FAC/ETAGES

STATE COMPONENTS

```
*Numero instance of INTEGER
Etat_porte instance of BOOLEAN
Appel_montee instance of BOOLEAN  → FAC/CONTROLE
Appel_descente instance of BOOLEAN  → FAC/CONTROLE
Panne_a_venir instance of BOOLEAN  → FAC/CONTROLE,
                                     FAC/REPARATEUR
```

ACTIONS

```
Detecter_panne
Verifier
*Mise_a_jour1
*Mise_a_jour2
*Mise_a_jour
```

BASIC CONSTRAINTS

INITIAL VALUATION

```
Panne_a_venir = FALSE
Etat_porte = FALSE
```

LOCAL CONSTRAINTS

STATE BEHAVIOUR

```
Appel_montee ⇒ ◊ Etat_porte
Appel_descente ⇒ ◊ Etat_porte
Etat_porte ⇒ ◊ ¬Etat_porte
Panne_a_venir ⇒ ◊ ¬Panne_a_venir
PremierEtage = self ⇔ Appel_descente = FALSE
DernierEtage = self ⇔ Appel_montee = FALSE
PremierEtage = self ⇔ Numero = -1
DernierEtage = self ⇔ Numero = 4
Numero >= -1
Numero <=4
```

EFFECTS OF ACTIONS

```
Detecter_panne : Panne_a_venir := TRUE
Reparateur.Reparer_etages : Panne_a_venir := FALSE
Utilisateurs.Appeler( MONTEE, _ ) : Appel_montee := TRUE
Utilisateurs.Appeler( DESCENTE, _ ) : Appel_descente := TRUE
Controle.Ouvrir_porte : Etat_porte := TRUE
Controle.Fermer_porte : Etat_porte := FALSE
Mise_a_jour1 : Appel_montee := FALSE
Mise_a_jour2 : Appel_descente := FALSE
```

CAPABILITY

```
F( Mise_a_jour1 | Controle.Phase <> UP )
F( Mise_a_jour2 | Controle.Phase <> DOWN )
```

Action Composition

Verifier ↔ Controle.Ouvrir_porte ; Mise_a_jour
Mise_a_jour ↔ Mise_a_jour1 ⊕ Mise_a_jour2 ⊕ DAC

Co-operation Constraints

Action Perception

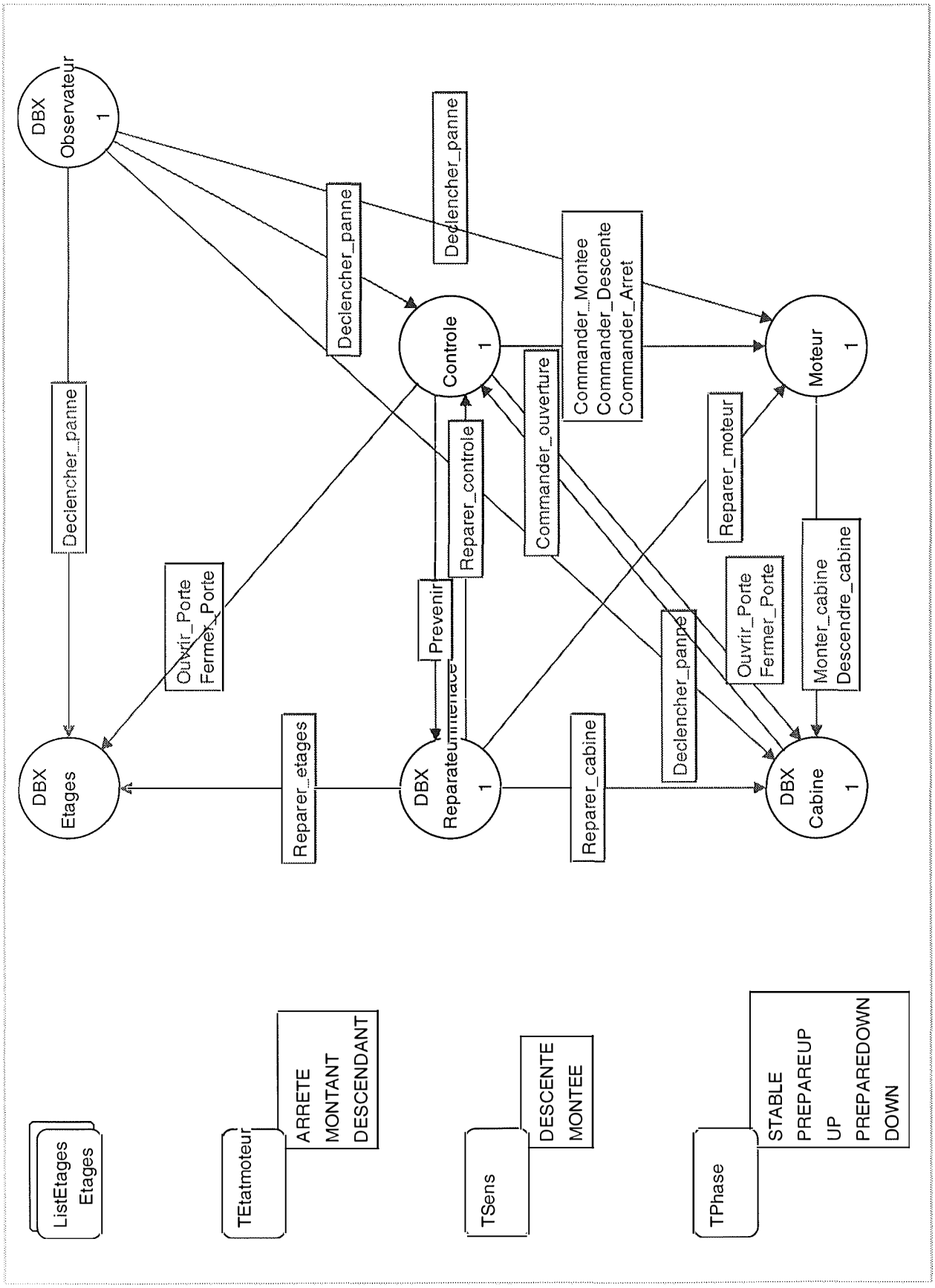
XK(Utilisateurs.Appeler(MONTEE,dest) with dest = self|
¬Appel_montee)
XK(Utilisateurs.Appeler(DESCENTE,dest) with dest = self|
¬Appel_descente)
XK(Controle.Ouvrir_porte | ¬Etat_porte)
XK(Controle.Fermer_porte | Etat_porte)
XK(Reparateur.Reparer_moteur | Panne_a_venir)

State Information

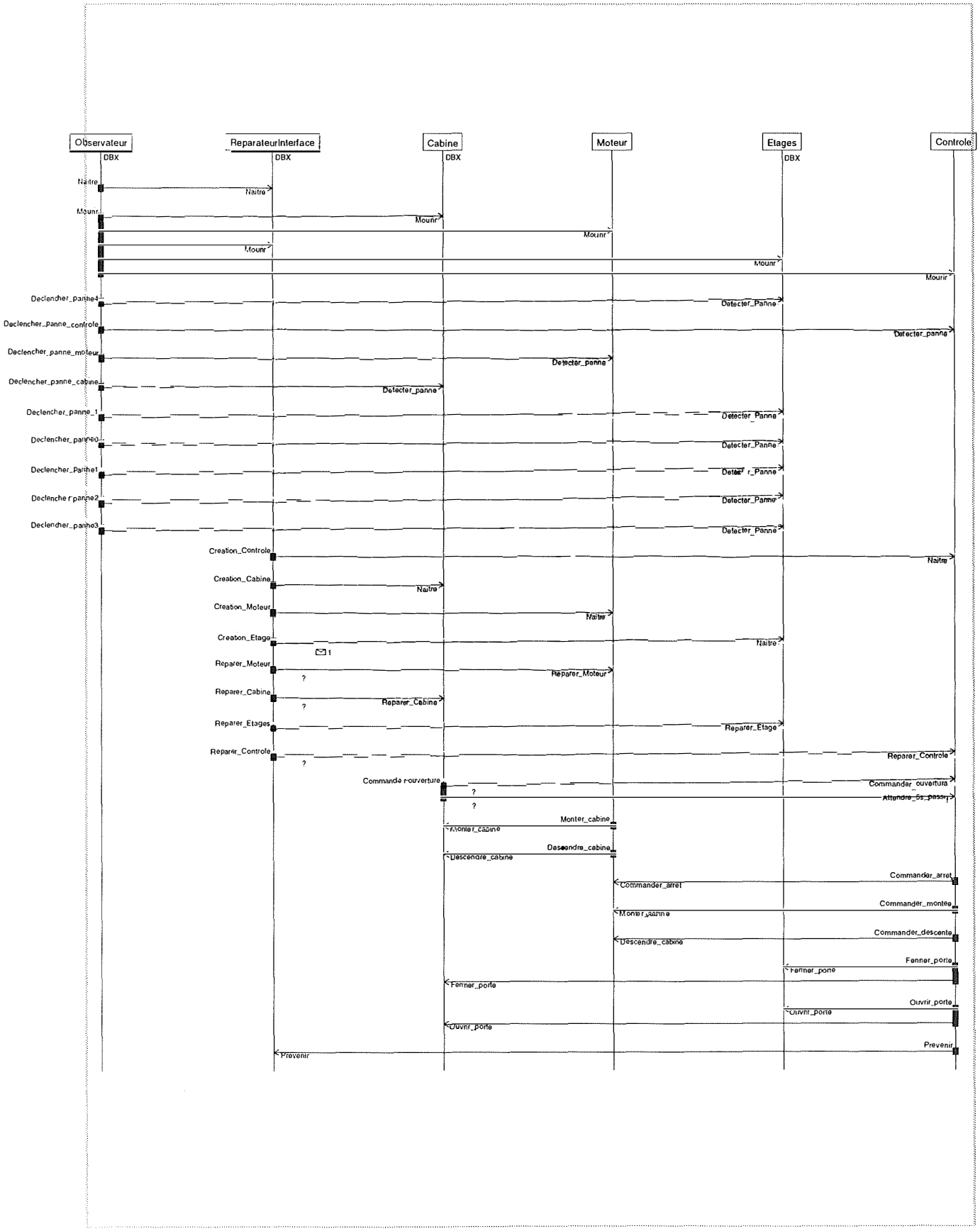
K(Panne_a_venir.X | TRUE)
K(Appel_montee.Controle | TRUE)
K(Appel_descente.Controle | TRUE)

SPECIFICATION DU SYSTEME EN OBLOG.

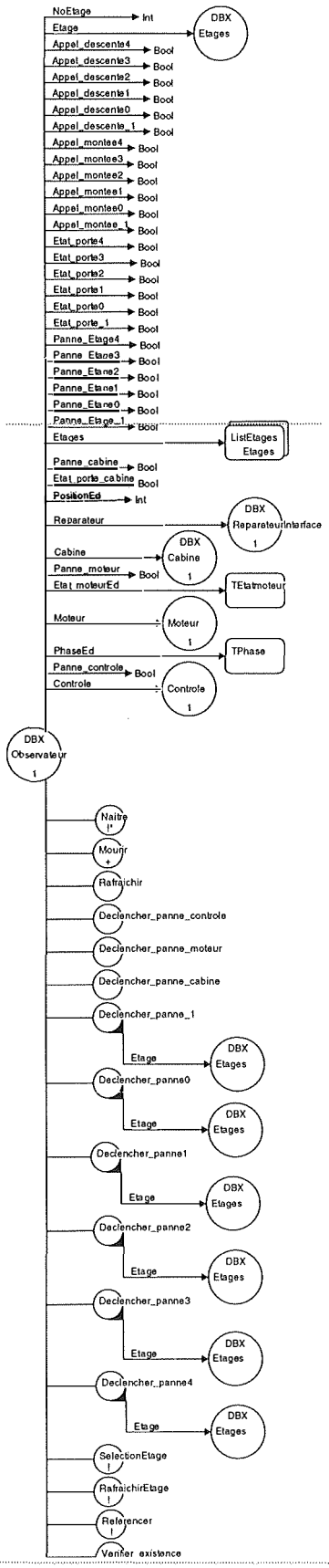
Contents: Community Diagram	Date: Wed Aug 28 1996 13:17:13
Unit: Ascenceur	By: Unknown



Contents: Unit Interaction Diagram **Unit:** Ascenceur	**Date:** Wed Aug 28 1996 13:17:13 **By:** Unknown



Contents: Declaration Diagram **Class:** Ascenceur:Observateur	**Date:** Wed Aug 28 1996 13:17:13 **By:** Unknown



Contents: Local Class Declaration	**Date:** Wed Aug 28 1996 13:17:13
Class: Observateur	**By:** Unknown

Single DBX Class `Observateur`

Interactions

to `Etages` **using**
- `Declencher_panne`

to `Controle` **using**
- `Declencher_panne`

to `Cabine` **using**
- `Declencher_panne`

to `Moteur` **using**
- `Declencher_panne`

Attributes

`Appel_descente0`	:	`Bool`
`Appel_descente1`	:	`Bool`
`Appel_descente2`	:	`Bool`
`Appel_descente3`	:	`Bool`
`Appel_descente4`	:	`Bool`
`Appel_descente_1`	:	`Bool`
`Appel_montee0`	:	`Bool`
`Appel_montee1`	:	`Bool`
`Appel_montee2`	:	`Bool`
`Appel_montee3`	:	`Bool`
`Appel_montee4`	:	`Bool`
`Appel_montee_1`	:	`Bool`
`Cabine`	:	`Cabine`
`Controle`	:	`Controle`
`Etage`	:	`Etages`
`Etages`	:	`ListEtages`
`Etat_moteurEd`	:	`TEtatmoteur`
`Etat_porte0`	:	`Bool`
`Etat_porte1`	:	`Bool`
`Etat_porte2`	:	`Bool`
`Etat_porte3`	:	`Bool`
`Etat_porte4`	:	`Bool`
`Etat_porte_1`	:	`Bool`
`Etat_porte_cabine`	:	`Bool`
`Moteur`	:	`Moteur`
`NoEtage`	:	`Int`
`Panne_Etage0`	:	`Bool`
`Panne_Etage1`	:	`Bool`
`Panne_Etage2`	:	`Bool`
`Panne_Etage3`	:	`Bool`
`Panne_Etage4`	:	`Bool`
`Panne_Etage_1`	:	`Bool`
`Panne_cabine`	:	`Bool`
`Panne_controle`	:	`Bool`
`Panne_moteur`	:	`Bool`
`PhaseEd`	:	`TPhase`
`PositionEd`	:	`Int`
`Reparateur`	:	`ReparateurInterface`

Operations

`*!Naitre`

```
Declencher_panne0
   Etage                                : Etages

Declencher_panne1
   Etage                                : Etages

Declencher_panne2
   Etage                                : Etages

Declencher_panne3
   Etage                                : Etages

Declencher_panne4
   Etage                                : Etages

Declencher_panne_1
   Etage                                : Etages

Declencher_panne_cabine
Declencher_panne_controle
Declencher_panne_moteur
Rafraichir
!RafraichirEtage
!Referencer
!SelectionEtage
!Verifier_existence
+Mourir
```

End Class Observateur

Operation Naitre

Calls

ReparateurInterface.Naitre **of object identified by** Reparateur

Operation Declencher_panne0

Parameters

Etage : Etages

Calls

Etages.Detecter_Panne **of object identified by**
Declencher_panne0.Etage

Operation Declencher_panne1

Parameters

Etage : Etages

Calls

Etages.Detecter_Panne **of object identified by**
Declencher_panne1.Etage

Operation Declencher_panne2

Parameters

Etage : Etages

Calls

Etages.Detecter_Panne **of object identified by**
Declencher_panne2.Etage

Operation Declencher_panne3

Parameters

Etage : Etages

Calls

Etages.Detecter_Panne **of object identified by** Declencher_panne3.Etage

Operation Declencher_panne4

Parameters

Etage : Etages

Calls

Etages.Detecter_Panne **of object identified by** Declencher_panne4.Etage

Operation Declencher_panne_1

Parameters

Etage : Etages

Calls

Etages.Detecter_Panne **of object identified by** Declencher_panne_1.Etage

Operation Declencher_panne_cabine

Calls

Cabine.Detecter_panne **of object identified by** Cabine

Operation Declencher_panne_controle

Calls

Controle.Detecter_panne **of object identified by** Controle

Operation Declencher_panne_moteur

Calls

Moteur.Detecter_panne **of object identified by** Moteur

Operation Rafraichir

Updates

```
NoEtage     :=  -1
Panne_cabine   :=  Cabine.Panne_a_venir
Panne_moteur   :=  Moteur.Panne_a_venir
Panne_controle     :=  Controle.Panne_a_venir
Etat_porte_cabine     :=  Cabine.Etat_Porte
PositionEd     :=  Cabine.Position.Numero
Etat_moteurEd  :=  Moteur.Etat
PhaseEd     :=  Controle.Phase
```

Operation RafraichirEtage

Updates

```
Etat_porte_1   :=
    if(NoEtage = -1,Etage.Etat_Porte, Etat_porte_1)
Etat_porte0    :=
    if(NoEtage = 0,Etage.Etat_Porte, Etat_porte0)
Etat_porte1    :=
    if(NoEtage = 1,Etage.Etat_Porte, Etat_porte1)
Etat_porte2    :=
    if(NoEtage = 2,Etage.Etat_Porte, Etat_porte2)
Etat_porte3    :=
    if(NoEtage = 3,Etage.Etat_Porte, Etat_porte3)
Etat_porte4    :=
    if(NoEtage = 4,Etage.Etat_Porte, Etat_porte4)
NoEtage     :=  NoEtage + 1
Panne_Etage_1  :=
    if(NoEtage = -1,Etage.Panne_a_venir, Panne_Etage_1)
Panne_Etage0   :=
    if(NoEtage = 0,Etage.Panne_a_venir, Panne_Etage0)
Panne_Etage1   :=
    if(NoEtage = 1,Etage.Panne_a_venir, Panne_Etage1)
Panne_Etage2   :=
    if(NoEtage = 2,Etage.Panne_a_venir, Panne_Etage2)
Panne_Etage3   :=
```

```
    if(NoEtage = 3,Etage.Panne_a_venir, Panne_Etage3)
Panne_Etage4   :=
    if(NoEtage = 4,Etage.Panne_a_venir, Panne_Etage4)
Appel_montee_1    :=
    if(NoEtage = -1,Etage.Appel_montee, Appel_montee_1)
Appel_montee0  :=
    if(NoEtage = 0,Etage.Appel_montee, Appel_montee0)
Appel_montee1  :=
    if(NoEtage = 1,Etage.Appel_montee, Appel_montee1)
Appel_montee2  :=
    if(NoEtage = 2,Etage.Appel_montee, Appel_montee2)
Appel_montee3  :=
    if(NoEtage = 3,Etage.Appel_montee, Appel_montee3)
Appel_montee4  :=
    if(NoEtage = 4,Etage.Appel_montee, Appel_montee4)
Appel_descente_1  :=
    if(NoEtage = -1,Etage.Appel_descente, Appel_descente_1)
Appel_descente0   :=
    if(NoEtage = 0,Etage.Appel_descente, Appel_descente0)
Appel_descente1   :=
    if(NoEtage = 1,Etage.Appel_descente, Appel_descente1)
Appel_descente2   :=
    if(NoEtage = 2,Etage.Appel_descente, Appel_descente2)
Appel_descente3   :=
    if(NoEtage = 3,Etage.Appel_descente, Appel_descente3)
Appel_descente4   :=
    if(NoEtage = 4,Etage.Appel_descente, Appel_descente4)
```

Operation Referencer

Updates

```
Controle    :=  ONE [ Controle | True ]
Cabine  :=  ONE [ Cabine | True ]
Moteur  :=  ONE [ Moteur | True ]
Etages  :=  ALL [ Etages | True ]
```

Operation SelectionEtage

Updates

```
Etage   :=  ONE [ Etages | Numero = self.NoEtage ]
```

Operation Verifier_existence

No Properties

Operation Mourir

Calls

Moteur.Mourir **of object identified by** Moteur

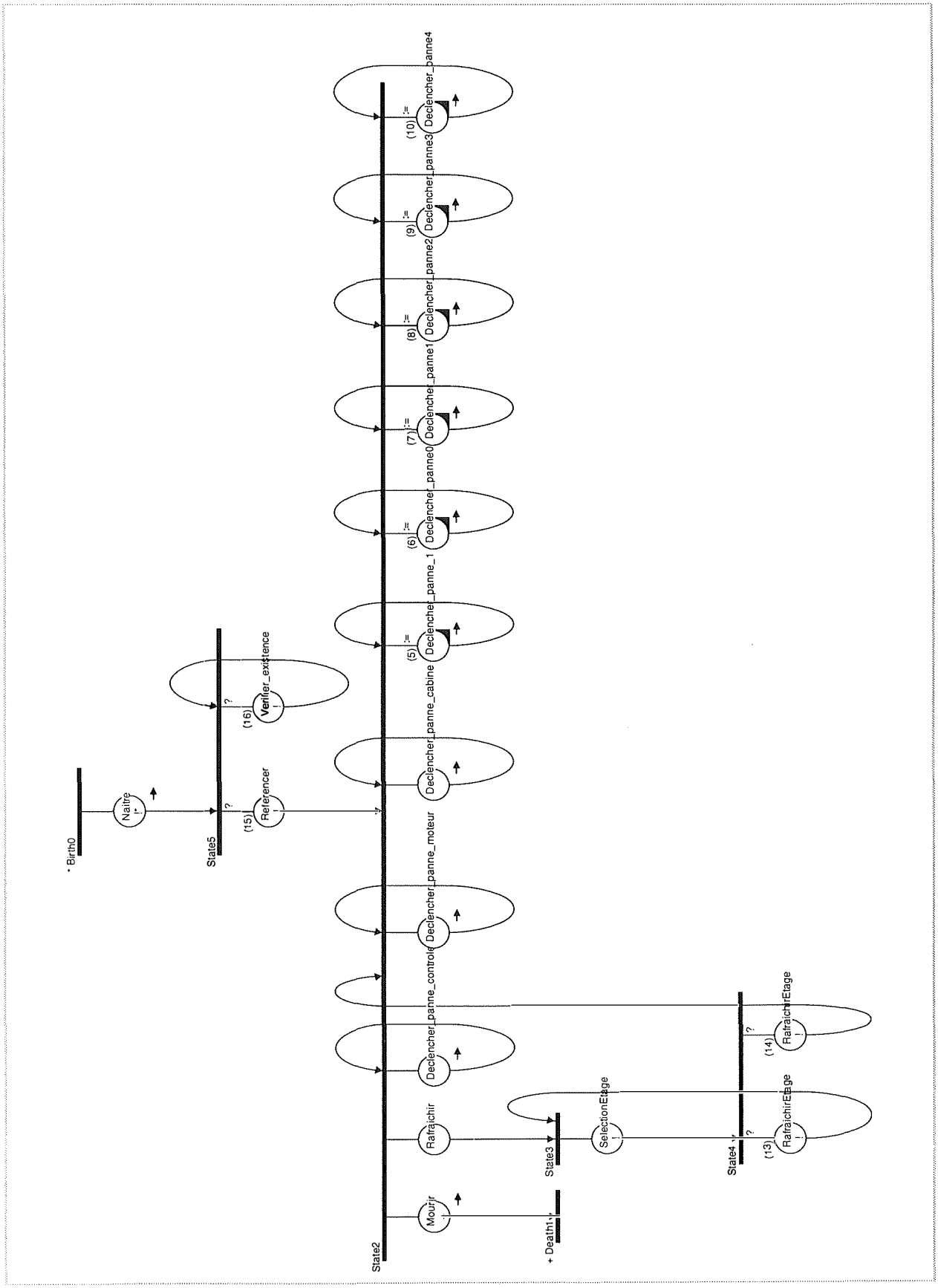
ReparateurInterface.Mourir **of object identified by** Reparateur

Etages.Mourir **of each object in** Etages

Controle.Mourir **of object identified by** Controle

Cabine.Mourir **of object identified by** Cabine

Contents:	Behavior Diagram	**Date:**	Wed Aug 28 1996 13:17:13
Class:	Ascenceur:Observateur	**By:**	Unknown



Behavior of class `Observateur`

Birth State `Birth0` **:**

⟶▷ `*!Naitre` ⟶▷ `State5`

State `State2` **:**

⟶▷ `Rafraichir` ⟶▷ `State3`

⟶▷ `Declencher_panne_controle` ⟶▷ `State2`

⟶▷ `Declencher_panne_moteur` ⟶▷ `State2`

⟶▷ `Declencher_panne_cabine` ⟶▷ `State2`

⟶▷ `Declencher_panne_1` ⟶▷ `State2`

```
Declencher_panne_1.Etage:= ONE[ Etages | Numero = -1 ]
```

⟶▷ `Declencher_panne0` ⟶▷ `State2`

```
Declencher_panne0.Etage:= ONE[ Etages | Numero = 0 ]
```

⟶▷ `Declencher_panne1` ⟶▷ `State2`

```
Declencher_panne1.Etage:= ONE[ Etages | Numero = 1 ]
```

⟶▷ `Declencher_panne2` ⟶▷ `State2`

```
Declencher_panne2.Etage:= ONE[ Etages | Numero = 2 ]
```

⟶▷ `Declencher_panne3` ⟶▷ `State2`

```
Declencher_panne3.Etage:= ONE[ Etages | Numero = 3 ]
```

⟶▷ `Declencher_panne4` ⟶▷ `State2`

```
Declencher_panne4.Etage:= ONE[ Etages | Numero = 4 ]
```

⟶▷ `+Mourir` ⟶▷ `Death1`

State `State3` **:**

⟶▷ `!SelectionEtage` ⟶▷ `State4`

State `State4` **:**

⟶▷ `!RafraichirEtage` ⟶▷ `State3`

?: `NoEtage<4`

⟶▷ `!RafraichirEtage` ⟶▷ `State2`

?: `NoEtage = 4`

State State5 **:**

——▷ !Referencer ——▷ State2
?: EXISTS [Moteur | TRUE]

——▷ !Verifier_existence ——▷ State5
?: not EXISTS [Moteur | TRUE]

Death State Death1 **:**

Expressions in behavior of class `Observateur`

Transition (6) with `Declencher_panne0`

```
Declencher_panne0.Etage:=  ONE[ Etages | Numero = 0 ]
```

Transition (7) with `Declencher_panne1`

```
Declencher_panne1.Etage:=  ONE[ Etages | Numero = 1 ]
```

Transition (8) with `Declencher_panne2`

```
Declencher_panne2.Etage:=  ONE[ Etages | Numero = 2 ]
```

Transition (9) with `Declencher_panne3`

```
Declencher_panne3.Etage:=  ONE[ Etages | Numero = 3 ]
```

Transition (10) with `Declencher_panne4`

```
Declencher_panne4.Etage:=  ONE[ Etages | Numero = 4 ]
```

Transition (5) with `Declencher_panne_1`

```
Declencher_panne_1.Etage:=  ONE[ Etages | Numero = -1 ]
```

Transition (13) with `RafraichirEtage`

?: `NoEtage<4`

Transition (14) with `RafraichirEtage`

?: `NoEtage = 4`

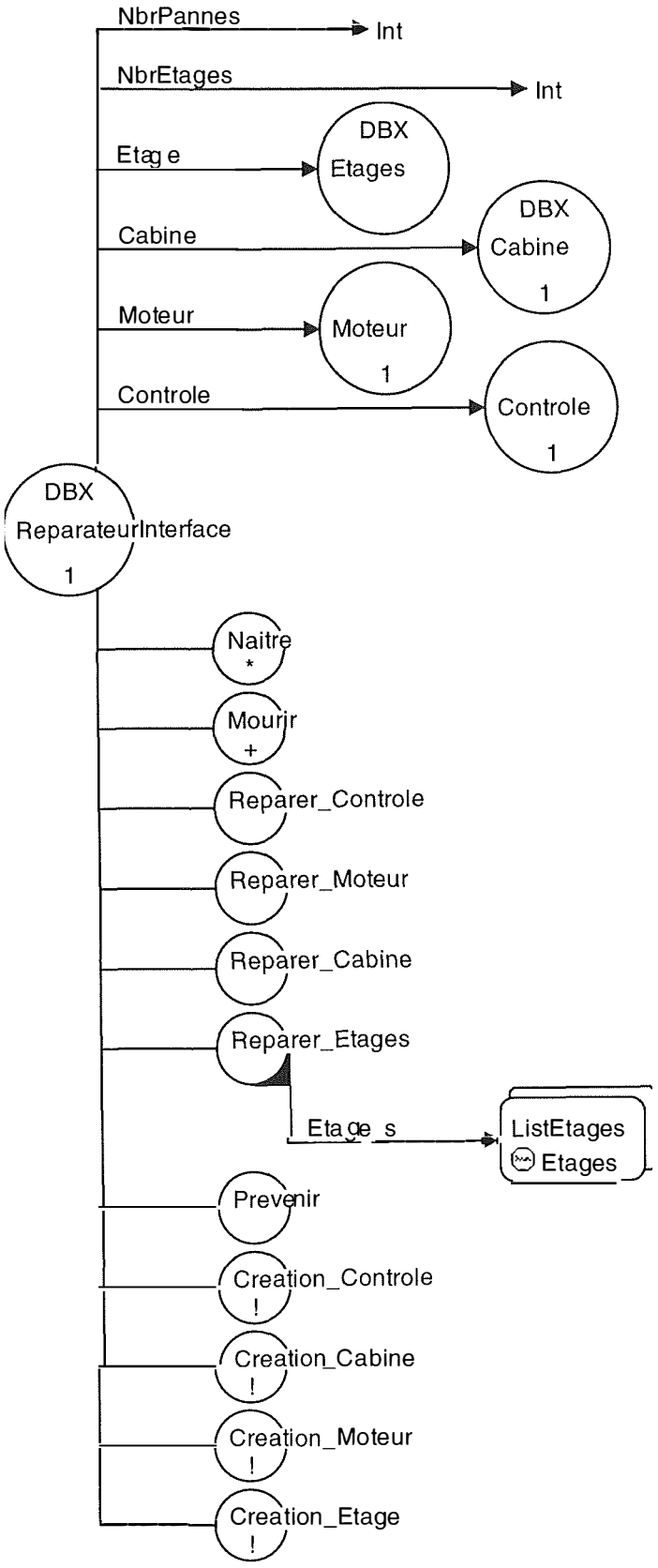
Transition (15) with `Referencer`

?: `EXISTS [ Moteur | TRUE ]`

Transition (16) with `Verifier_existence`

?: `not EXISTS [ Moteur | TRUE ]`

Contents: Declaration Diagram **Class:** Ascenceur:ReparateurInterface	**Date:** Wed Aug 28 1996 13:17:13 **By:** Unknown



Contents: Local Class Declaration	Date: Wed Aug 28 1996 13:17:13
Class: ReparateurInterface	By: Unknown

Single DBX Class ReparateurInterface

Interactions

to Controle **using**
- Reparer_controle

to Cabine **using**
- Reparer_cabine

to Etages **using**
- Reparer_etages

to Moteur **using**
- Reparer_moteur

Attributes

Cabine	:	Cabine
Controle	:	Controle
Etage	:	Etages
Moteur	:	Moteur
NbrEtages	:	Int
NbrPannes	:	Int

Operations

*Naitre

Prevenir

Reparer_Cabine

Reparer_Controle

Reparer_Etages
Etages : ListEtages

Reparer_Moteur

!Creation_Cabine

!Creation_Controle

!Creation_Etage

!Creation_Moteur

+Mourir

End Class ReparateurInterface

Operation Naitre

Updates

NbrEtages := 6

Operation Prevenir

Updates

Etage :=
if(EXISTS[Etages | Panne_a_venir],ONE [Etages | Panne_a_venir],ONE [Etages | TRUE])

Operation Reparer_Cabine

Calls

Cabine.Reparer_Cabine **of object identified by** Cabine
?: Cabine.Panne_a_venir

Operation Reparer_Controle

Calls

Controle.Reparer_Controle **of object identified by** Controle
?: Controle.Panne_a_venir

Operation Reparer_Etages

Parameters

Etages : ListEtages

Calls

Etages.Reparer_Etage **of each object in** Reparer_Etages.Etages

Operation Reparer_Moteur

Calls

Moteur.Reparer_Moteur **of object identified by** Moteur
?: Moteur.Panne_a_venir

Operation Creation_Cabine

Calls

Cabine.Naitre **of object identified by** Cabine

Operation Creation_Controle

Calls

Controle.Naitre **of object identified by** Controle

Operation Creation_Etage

Updates

```
NbrEtages := NbrEtages - 1
```

Calls

Etages.Naitre **of object identified by** Etage
with instantiations

```
Naitre.Numero := NbrEtages
```

Operation Creation_Moteur

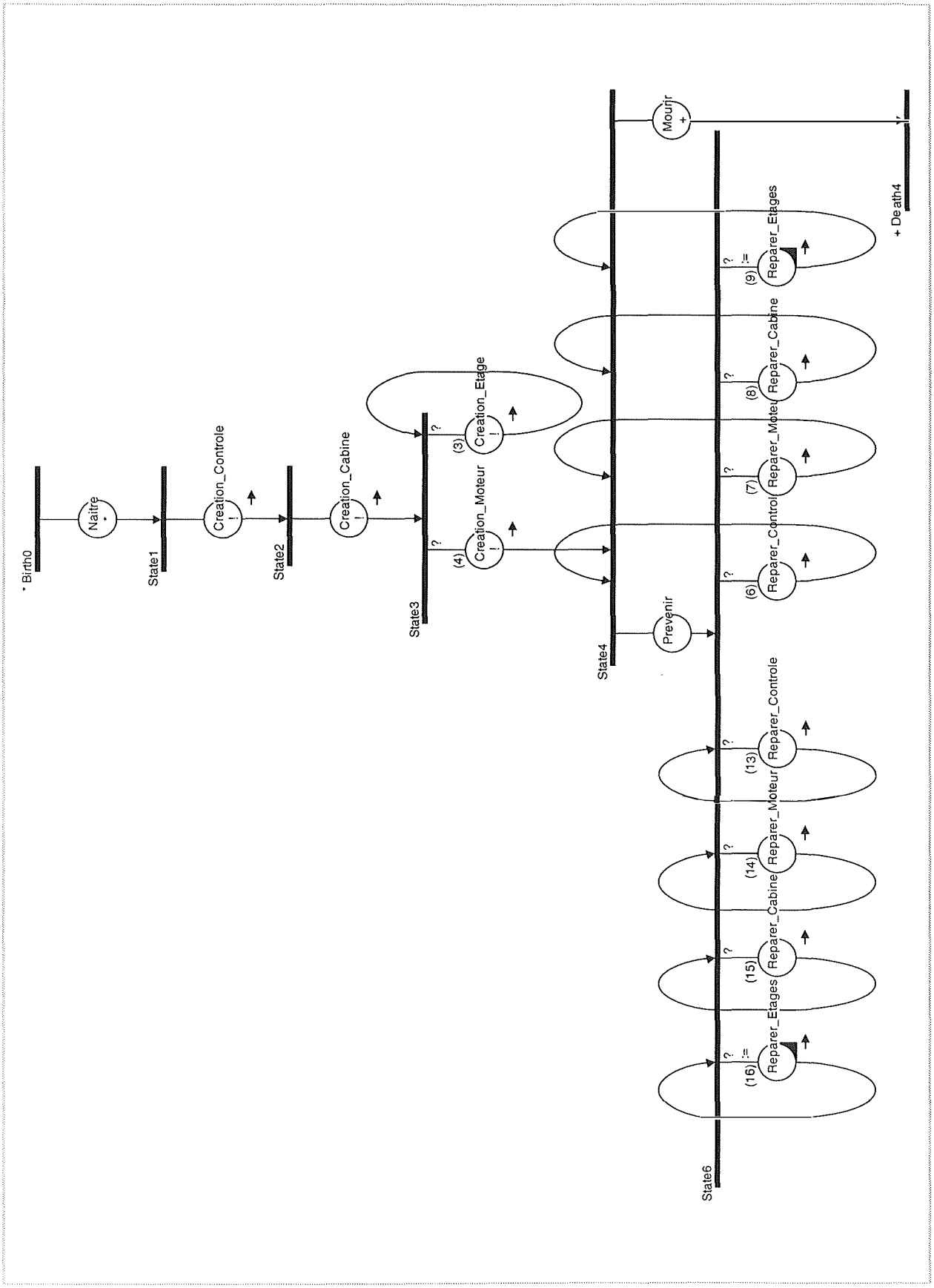
Calls

Moteur.Naitre **of object identified by** Moteur

Operation Mourir

No Properties

Contents: Behavior Diagram	**Date:** Wed Aug 28 1996 13:17:13
Class: Ascenceur:ReparateurInterface	**By:** Unknown



Behavior of class ReparateurInterface

Birth State Birth0 **:**

⟶ *Naitre ⟶ State1

State State1 **:**

⟶ !Creation_Controle ⟶ State2

State State2 **:**

⟶ !Creation_Cabine ⟶ State3

State State3 **:**

⟶ !Creation_Etage ⟶ State3
?: NbrEtages>0

⟶ !Creation_Moteur ⟶ State4
?: NbrEtages = 0

State State4 **:**

⟶ +Mourir ⟶ Death4

⟶ Prevenir ⟶ State6

State State6 **:**

⟶ Reparer_Controle ⟶ State6
?:
Controle.Panne_a_venir and (EXISTS[Etages | Panne_a_venir]
or Moteur.Panne_a_venir or Cabine.Panne_a_venir)

⟶ Reparer_Moteur ⟶ State6
?:
Moteur.Panne_a_venir and (EXISTS[Etages | Panne_a_venir] or
Controle.Panne_a_venir or Cabine.Panne_a_venir)

⟶ Reparer_Cabine ⟶ State6
?:
Cabine.Panne_a_venir and (EXISTS[Etages | Panne_a_venir] or
Moteur.Panne_a_venir or Controle.Panne_a_venir)

⟶▷ Reparer_Etages ⟶▷ State6

?:

EXISTS[Etages | Panne_a_venir] and (Controle.Panne_a_venir or Moteur.Panne_a_venir or Cabine.Panne_a_venir)

Reparer_Etages.Etages:= ALL[Etages | Panne_a_venir]

⟶▷ Reparer_Controle ⟶▷ State4

?:

Controle.Panne_a_venir and not Moteur.Panne_a_venir and not Cabine.Panne_a_venir and not EXISTS[Etages | Panne_a_venir]

⟶▷ Reparer_Moteur ⟶▷ State4

?:

Moteur.Panne_a_venir and not Controle.Panne_a_venir and not Cabine.Panne_a_venir and not EXISTS[Etages | Panne_a_venir]

⟶▷ Reparer_Cabine ⟶▷ State4

?:

Cabine.Panne_a_venir and not Controle.Panne_a_venir and not Moteur.Panne_a_venir and not EXISTS[Etages | Panne_a_venir]

⟶▷ Reparer_Etages ⟶▷ State4

?:

EXISTS[Etages | Panne_a_venir] and not Controle.Panne_a_venir and not Moteur.Panne_a_venir and not Cabine.Panne_a_venir

Reparer_Etages.Etages:= ALL[Etages | Panne_a_venir]

Death State Death4 **:**

Expressions in behavior of class `ReparateurInterface`

Transition (8) with `Reparer_Cabine`

?:

```
Cabine.Panne_a_venir and not Controle.Panne_a_venir and not M
oteur.Panne_a_venir and not EXISTS[ Etages | Panne_a_venir ]
```

Transition (15) with `Reparer_Cabine`

?:

```
Cabine.Panne_a_venir and (EXISTS[ Etages | Panne_a_venir ] or
 Moteur.Panne_a_venir or Controle.Panne_a_venir)
```

Transition (13) with `Reparer_Controle`

?:

```
Controle.Panne_a_venir and (EXISTS[ Etages | Panne_a_venir ]
or Moteur.Panne_a_venir or Cabine.Panne_a_venir)
```

Transition (6) with `Reparer_Controle`

?:

```
Controle.Panne_a_venir and not Moteur.Panne_a_venir and not C
abine.Panne_a_venir and not EXISTS[ Etages | Panne_a_venir ]
```

Transition (9) with `Reparer_Etages`

?:

```
EXISTS[ Etages | Panne_a_venir ] and not Controle.Panne_a_ven
ir and not Moteur.Panne_a_venir and not Cabine.Panne_a_venir
```

```
Reparer_Etages.Etages:=  ALL[Etages | Panne_a_venir]
```

Transition (16) with `Reparer_Etages`

?:

```
EXISTS[ Etages | Panne_a_venir ] and (Controle.Panne_a_venir
or Moteur.Panne_a_venir or Cabine.Panne_a_venir)
```

```
Reparer_Etages.Etages:=  ALL[Etages | Panne_a_venir]
```

Transition (14) with `Reparer_Moteur`

?:

```
Moteur.Panne_a_venir and (EXISTS[ Etages | Panne_a_venir ] or
 Controle.Panne_a_venir or Cabine.Panne_a_venir)
```

Transition (7) with `Reparer_Moteur`

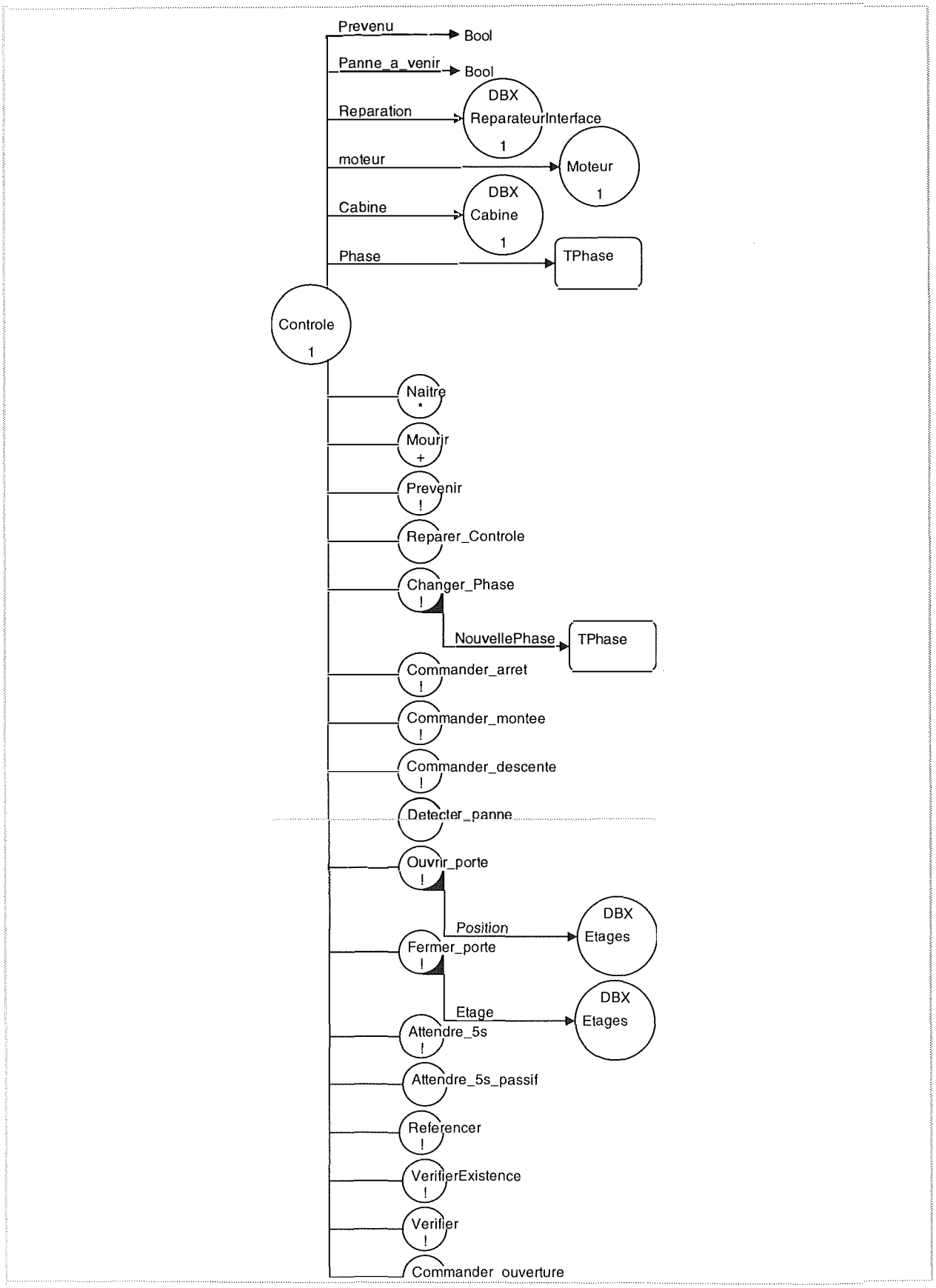
?:

```
Moteur.Panne_a_venir and not Controle.Panne_a_venir and not C
abine.Panne_a_venir and not EXISTS[ Etages | Panne_a_venir ]
```

Transition (3) with `Creation_Etage`
?: `NbrEtages>0`

Transition (4) with `Creation_Moteur`
?: `NbrEtages = 0`

Contents: Declaration Diagram **Class:** Ascenceur:Controle	**Date:** Wed Aug 28 1996 13:17:13 **By:** Unknown



Contents: Local Class Declaration	Date: Wed Aug 28 1996 13:17:13
Class: Controle	By: Unknown

Single Class Controle

Interactions

to Cabine **using**
- Ouvrir_Porte
- Fermer_Porte

to Etages **using**
- Ouvrir_Porte
- Fermer_Porte

to ReparateurInterface **using**
- Prevenir

to Moteur **using**
- Commander_Montee
- Commander_Descente
- Commander_Arret

Attributes

Cabine	:	Cabine
Panne_a_venir	:	Bool
Phase	:	TPhase
Prevenu	:	Bool
Reparation	:	ReparateurInterface
moteur	:	Moteur

Operations

*Naitre
Attendre_5s_passif
Commander_ouverture
Detecter_panne
Reparer_Controle
!Attendre_5s
!Changer_Phase
   NouvellePhase : TPhase

!Commander_arret
!Commander_descente
!Commander_montee
!Fermer_porte
   Etage : Etages

!Ouvrir_porte
   Position : Etages

!Prevenir
!Referencer
!Verifier
!VerifierExistence
+Mourir

End Class `Controle`

Operation Naitre

Updates

```
Phase   := TPhase$STABLE
Panne_a_venir := FALSE
Prevenu    := FALSE
```

Operation Attendre_5s_passif

No Properties

Operation Commander_ouverture

No Properties

Operation Detecter_panne

Updates

```
Panne_a_venir := TRUE
```

Operation Reparer_Controle

Updates

```
Panne_a_venir := FALSE
```

Operation Attendre_5s

No Properties

Operation Changer_Phase

Parameters

NouvellePhase : TPhase

Updates

Phase := Changer_Phase.NouvellePhase

Operation Commander_arret

Calls

Moteur.Commander_arret **of object identified by** moteur

Operation Commander_descente

Calls

Moteur.Descendre_cabine **of object identified by** moteur

Operation Commander_montee

Calls

Moteur.Monter_cabine **of object identified by** moteur

Operation Fermer_porte

Parameters

Etage : Etages

Calls

Cabine.Fermer_porte **of object identified by** Cabine

Etages.Fermer_porte **of object identified by** Fermer_porte.Etage

Operation Ouvrir_porte

Parameters

Position : Etages

Calls

Cabine.Ouvrir_porte **of object identified by** Cabine

Etages.Ouvrir_porte **of object identified by** Ouvrir_porte.Position

Operation Prevenir

Updates

Prevenu := TRUE

Calls

ReparateurInterface.Prevenir **of object identified by** Reparation

Operation Referencer

Updates

moteur := ONE[Moteur | TRUE]
Cabine := ONE[Cabine | TRUE]
Reparation := ONE[ReparateurInterface | TRUE]

Operation Verifier

Updates

Prevenu := FALSE

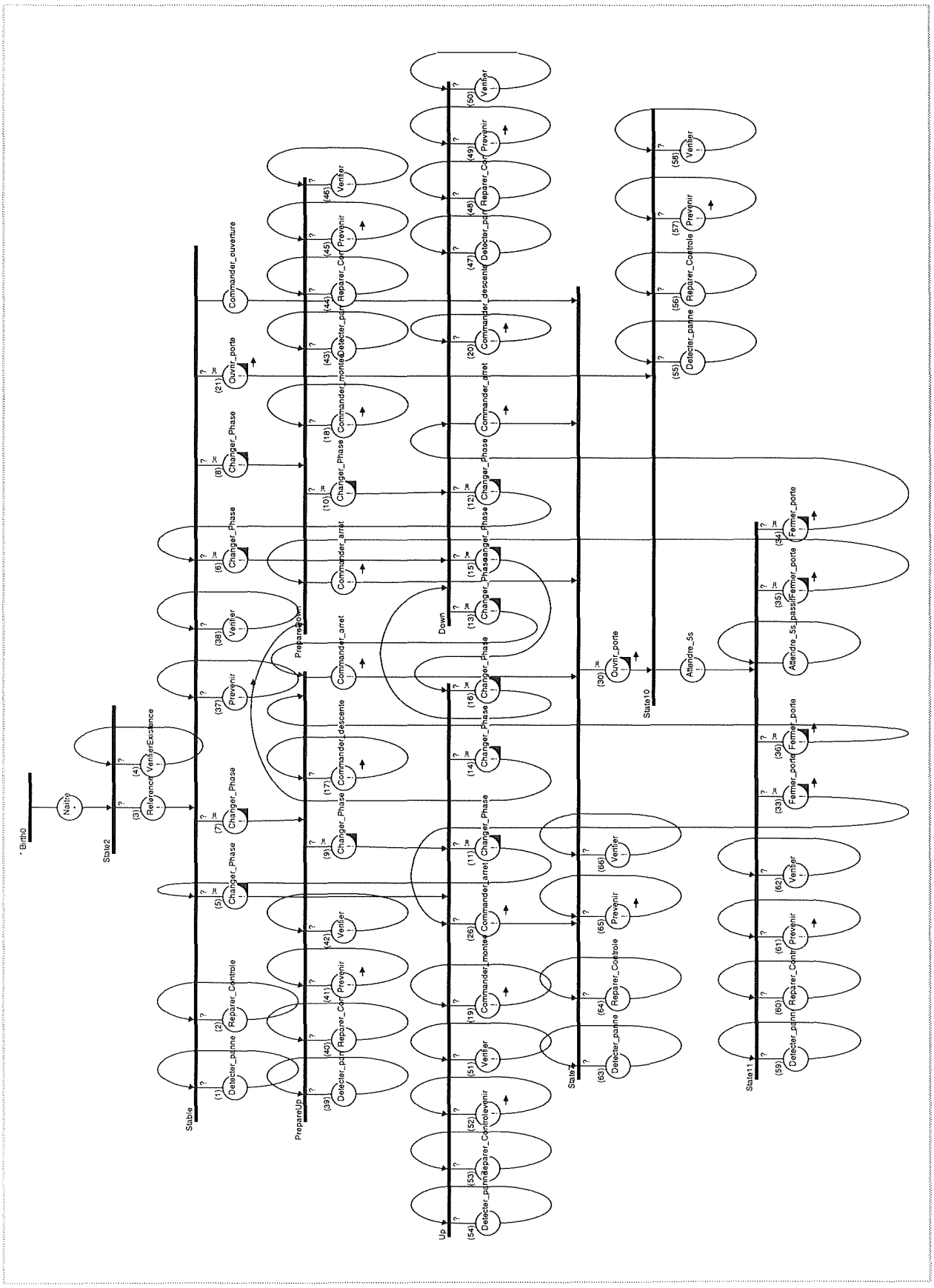
Operation VerifierExistence

No Properties

Operation Mourir

No Properties

Contents: Behavior Diagram	**Date:** Wed Aug 28 1996 13:17:13
Class: Ascenceur:Controle	**By:** Unknown



Behavior of class Controle

Birth State Birth0 **:**

⟶ *Naitre ⟶ State2

State Down **:**

⟶ Detecter_panne ⟶ Down
?: not Panne_a_venir

⟶ Reparer_Controle ⟶ Down
?: Panne_a_venir

⟶ !Prevenir ⟶ Down
?:
(Panne_a_venir or Cabine.Panne_a_venir or moteur.Panne_a_veni
r or EXISTS[Etages | Panne_a_venir]) and not Prevenu

⟶ !Verifier ⟶ Down
?:
not Panne_a_venir and not Cabine.Panne_a_venir and not moteur
.Panne_a_venir and not EXISTS[Etages | Panne_a_venir] and Pre
venu

⟶ !Commander_arret ⟶ State7

⟶ !Commander_descente ⟶ Down
?:
not Cabine.Position.Appel_descente and not isin(Cabine.Etages
_Arret,Cabine.Position)

⟶ !Changer_Phase ⟶ Up
?:
not EXISTS[Etages | self.Cabine.Position.Numero > Numero and
(Appel_montee or Appel_descente or isin(Self.Cabine.Etages_A
rret,current))] and EXISTS[Etages | self.Cabine.Position.Num
ero < Numero and Appel_montee] and not Cabine.Position.Appel
_descente and not isin(Cabine.Etages_Arret,Cabine.Position)
Changer_Phase.NouvellePhase:= TPhase$UP

⟶ !Changer_Phase ⟶ PrepareUp
?:
EXISTS [Etages | self.Cabine.Position.Numero > Numero and Ap
pel_montee] and not EXISTS[Etages | self.Cabine.Position.Nu
mero > Numero and (Appel_descente or isin(self.Cabine.Etages_
Arret,current))] and not Cabine.Position.Appel_montee and not
isin(Cabine.Etages_Arret,Cabine.Position)
Changer_Phase.NouvellePhase:= TPhase$PREPAREUP

⟶ !Changer_Phase ⟶ Stable
?:

```
not EXISTS[Etages | Appel_montee or Appel_descente or isin(se
lf.Cabine.Etages_Arret, current)] or (EXISTS[Etages | self.Ca
bine.Position.Numero < Numero and Appel_descente] and not EXI
STS[Etages| Appel_montee or isin(self.Cabine.Etages_Arret,cur
rent)] and not EXISTS[Etages| self.Cabine.Position.Numero > N
umero and Appel_descente] and not Cabine.Position.Appel_desce
nte)
Changer_Phase.NouvellePhase:= TPhase$STABLE
```

State PrepareDown :

⟶ Detecter_panne ⟹ PrepareDown
?: not Panne_a_venir

⟶ Reparer_Controle ⟶ PrepareDown
?: Panne_a_venir

⟶ !Commander_arret ⟶ State7

⟶ !Prevenir ⟶ PrepareDown
?:

```
(Panne_a_venir or Cabine.Panne_a_venir or moteur.Panne_a_veni
r or EXISTS[Etages | Panne_a_venir]) and not Prevenu
```

⟶ !Verifier ⟶ PrepareDown
?:

```
not Panne_a_venir and not Cabine.Panne_a_venir and not moteur
.Panne_a_venir and not EXISTS[Etages | Panne_a_venir] and Pre
venu
```

⟶ !Commander_montee ⟶ PrepareDown
?:

```
EXISTS[Etages | self.Cabine.Position.Numero < Numero and Appe
l_descente]
```

⟶ !Changer_Phase ⟶ Down
?:

```
not EXISTS[Etages | self.Cabine.Position.Numero < Numero and
Appel_descente]
Changer_Phase.NouvellePhase:= TPhase$DOWN
```

State PrepareUp :

⟶ Detecter_panne ⟹ PrepareUp
?: not Panne_a_venir

⟶ Reparer_Controle ⟶ PrepareUp
?: Panne_a_venir

⟶ !Prevenir ⟶ PrepareUp

?:
(Panne_a_venir or Cabine.Panne_a_venir or moteur.Panne_a_veni
r or EXISTS[Etages | Panne_a_venir]) and not Prevenu

——▷ !Verifier ——▷ PrepareUp

?:
not Panne_a_venir and not Cabine.Panne_a_venir and not moteur
.Panne_a_venir and not EXISTS[Etages | Panne_a_venir] and Pre
venu

——▷ !Commander_arret ——▷ State7

——▷ !Commander_descente ——▷ PrepareUp

?:
EXISTS[Etages | self.Cabine.Position.Numero > Numero and Appe
l_montee]

——▷ !Changer_Phase ——▷ Up

?:
not EXISTS[Etages | self.Cabine.Position.Numero > Numero and
Appel_montee]

Changer_Phase.NouvellePhase:= TPhase$UP

State Stable **:**

——▷ Commander_ouverture ——▷ State7

——▷ !Prevenir ——▷ Stable

?:
(Panne_a_venir or Cabine.Panne_a_venir or moteur.Panne_a_veni
r or EXISTS[Etages | Panne_a_venir]) and not Prevenu

——▷ !Verifier ——▷ Stable

?:
not Panne_a_venir and not Cabine.Panne_a_venir and not moteur
.Panne_a_venir and not EXISTS[Etages | Panne_a_venir] and Pre
venu

——▷ !Ouvrir_porte ——▷ State10

?:
Cabine.Position.Appel_montee or Cabine.Position.Appel_descent
e

Ouvrir_porte.Position:= Cabine.Position

——▷ !Changer_Phase ——▷ Down

?:
EXISTS[Etages | self.Cabine.Position.Numero > Numero and (App
el_descente or isin(self.Cabine.Etages_Arret,current))]

Changer_Phase.NouvellePhase:= TPhase$DOWN

——▷ !Changer_Phase ——▷ PrepareUp

?:

```
      EXISTS[Etages | self.Cabine.Position.Numero > Numero and Appe
      l_montee ]
   Changer_Phase.NouvellePhase:=  TPhase$PREPAREUP

--> !Changer_Phase --> PrepareDown
   ?:
      EXISTS[Etages | self.Cabine.Position.Numero < Numero and Appe
      l_descente ]
   Changer_Phase.NouvellePhase:=  TPhase$PREPAREDOWN

--> !Changer_Phase --> Up
   ?:
      EXISTS[Etages | self.Cabine.Position.Numero < Numero and (App
      el_montee or isin(self.Cabine.Etages_Arret,current))]
   Changer_Phase.NouvellePhase:=  TPhase$UP

--> Detecter_panne --> Stable
   ?: not Panne_a_venir

--> Reparer_Controle  --> Stable
   ?: Panne_a_venir
```

State State10 **:**

```
--> Detecter_panne --> State10
   ?: not Panne_a_venir

--> Reparer_Controle  --> State10
   ?: Panne_a_venir

--> !Prevenir  --> State10
   ?:
      (Panne_a_venir or Cabine.Panne_a_venir or moteur.Panne_a_veni
      r or EXISTS[Etages | Panne_a_venir]) and not Prevenu

--> !Verifier  --> State10
   ?:
      not Panne_a_venir and not Cabine.Panne_a_venir and not moteur
      .Panne_a_venir and not EXISTS[Etages | Panne_a_venir] and Pre
      venu

--> !Attendre_5s  --> State11
```

State State11 **:**

```
--> Detecter_panne --> State11
   ?: not Panne_a_venir

--> Reparer_Controle  --> State11
```

?: Panne_a_venir

⟶ !Prevenir ⟶ State11

?:
(Panne_a_venir or Cabine.Panne_a_venir or moteur.Panne_a_veni
r or EXISTS[Etages | Panne_a_venir]) and not Prevenu

⟶ !Verifier ⟶ State11

?:
not Panne_a_venir and not Cabine.Panne_a_venir and not moteur
.Panne_a_venir and not EXISTS[Etages | Panne_a_venir] and Pre
venu

⟶ !Fermer_porte ⟶ Up

?: Phase = TPhase$UP
Fermer_porte.Etage:= Cabine.Position

⟶ !Fermer_porte ⟶ Down

?: Phase = TPhase$DOWN
Fermer_porte.Etage:= Cabine.Position

⟶ !Fermer_porte ⟶ PrepareDown

?: Phase = TPhase$PREPAREDOWN
Fermer_porte.Etage:= Cabine.Position

⟶ !Fermer_porte ⟶ PrepareUp

?: Phase = TPhase$PREPAREUP
Fermer_porte.Etage:= Cabine.Position

⟶ Attendre_5s_passif ⟶ State11

State State2 **:**

⟶ !Referencer ⟶ Stable

?: EXISTS [Moteur | TRUE]

⟶ !VerifierExistence ⟶ State2

?: not EXISTS [Moteur | TRUE]

State State7 **:**

⟶ Detecter_panne ⟶ State7

?: not Panne_a_venir

⟶ Reparer_Controle ⟶ State7

?: Panne_a_venir

⟶ !Prevenir ⟶ State7

?:
(Panne_a_venir or Cabine.Panne_a_venir or moteur.Panne_a_veni
r or EXISTS[Etages | Panne_a_venir]) and not Prevenu

⟶ !Verifier ⟶ State7

?:
not Panne_a_venir and not Cabine.Panne_a_venir and not moteur
.Panne_a_venir and not EXISTS[Etages | Panne_a_venir] and Pre
venu

⟶ !Ouvrir_porte ⟶ State10

Ouvrir_porte.Position:= Cabine.Position

State Up **:**

⟶ !Verifier ⟶ Up

?:
not Panne_a_venir and not Cabine.Panne_a_venir and not moteur
.Panne_a_venir and not EXISTS[Etages | Panne_a_venir] and Pre
venu

⟶ !Prevenir ⟶ Up

?:
(Panne_a_venir or Cabine.Panne_a_venir or moteur.Panne_a_veni
r or EXISTS[Etages | Panne_a_venir]) and not Prevenu

⟶ Reparer_Controle ⟶ Up

?: Panne_a_venir

⟶ Detecter_panne ⟶ Up

?: not Panne_a_venir

⟶ !Commander_arret ⟶ State7

?:
Cabine.Position.Appel_montee or isin(Cabine.Etages_Arret,Cabi
ne.Position)

⟶ !Commander_montee ⟶ Up

?:
not Cabine.Position.Appel_montee and not isin(Cabine.Etages_A
rret,Cabine.Position)

⟶ !Changer_Phase ⟶ Down

?:
not EXISTS[Etages | self.Cabine.Position.Numero < Numero and
(Appel_montee or Appel_descente or isin(Self.Cabine.Etages_A
rret,current))] and EXISTS[Etages | self.Cabine.Position.Num
ero > Numero and Appel_descente] and not Cabine.Position.App
el_montee and not isin(Cabine.Etages_Arret,Cabine.Position)

Changer_Phase.NouvellePhase:= TPhase$DOWN

⟶▷ !Changer_Phase ⟶▷ PrepareDown

?:

```
EXISTS [ Etages | self.Cabine.Position.Numero < Numero and Ap
pel_descente ] and not EXISTS[ Etages | self.Cabine.Position.
Numero < Numero and (Appel_montee or isin(self.Cabine.Etages_
Arret,current))] and not Cabine.Position.Appel_descente and n
ot isin(Cabine.Etages_Arret,Cabine.Position)
```

```
Changer_Phase.NouvellePhase:=  TPhase$PREPAREDOWN
```

⟶▷ !Changer_Phase ⟶▷ Stable

?:

```
not EXISTS[Etages | Appel_montee or Appel_descente or isin(se
lf.Cabine.Etages_Arret, current)] or (EXISTS[Etages | self.Ca
bine.Position.Numero > Numero and Appel_montee] and not EXIST
S[Etages| Appel_descente or isin(self.Cabine.Etages_Arret,cur
rent)] and not EXISTS[Etages| self.Cabine.Position.Numero < N
umero and Appel_montee] and not Cabine.Position.Appel_montee)
```

```
Changer_Phase.NouvellePhase:=  TPhase$STABLE
```

Expressions in behavior of class `Controle`

Transition (1) with `Detecter_panne`
?: `not Panne_a_venir`

Transition (59) with `Detecter_panne`
?: `not Panne_a_venir`

Transition (55) with `Detecter_panne`
?: `not Panne_a_venir`

Transition (54) with `Detecter_panne`
?: `not Panne_a_venir`

Transition (47) with `Detecter_panne`
?: `not Panne_a_venir`

Transition (43) with `Detecter_panne`
?: `not Panne_a_venir`

Transition (39) with `Detecter_panne`
?: `not Panne_a_venir`

Transition (63) with `Detecter_panne`
?: `not Panne_a_venir`

Transition (2) with `Reparer_Controle`
?: `Panne_a_venir`

Transition (53) with `Reparer_Controle`
?: `Panne_a_venir`

Transition (44) with `Reparer_Controle`
?: `Panne_a_venir`

Transition (60) with `Reparer_Controle`
?: `Panne_a_venir`

Transition (40) with Reparer_Controle
?: Panne_a_venir

Transition (48) with Reparer_Controle
?: Panne_a_venir

Transition (64) with Reparer_Controle
?: Panne_a_venir

Transition (56) with Reparer_Controle
?: Panne_a_venir

Transition (15) with Changer_Phase
?:

```
not EXISTS[ Etages | self.Cabine.Position.Numero > Numero and
 (Appel_montee or Appel_descente or isin(Self.Cabine.Etages_A
rret,current))] and EXISTS[ Etages | self.Cabine.Position.Num
ero < Numero and Appel_montee ] and not Cabine.Position.Appel
_descente and not isin(Cabine.Etages_Arret,Cabine.Position)
```

```
Changer_Phase.NouvellePhase:=  TPhase$UP
```

Transition (16) with Changer_Phase
?:

```
not EXISTS[ Etages | self.Cabine.Position.Numero < Numero and
 (Appel_montee or Appel_descente or isin(Self.Cabine.Etages_A
rret,current))] and EXISTS[ Etages | self.Cabine.Position.Num
ero > Numero and Appel_descente ] and not Cabine.Position.App
el_montee and not isin(Cabine.Etages_Arret,Cabine.Position)
```

```
Changer_Phase.NouvellePhase:=  TPhase$DOWN
```

Transition (13) with Changer_Phase
?:

```
EXISTS [ Etages | self.Cabine.Position.Numero > Numero and Ap
pel_montee ] and not EXISTS[ Etages | self.Cabine.Position.Nu
mero > Numero and (Appel_descente or isin(self.Cabine.Etages_
Arret,current))] and not Cabine.Position.Appel_montee and not
 isin(Cabine.Etages_Arret,Cabine.Position)
```

```
Changer_Phase.NouvellePhase:=  TPhase$PREPAREUP
```

Transition (14) with Changer_Phase
?:

```
EXISTS [ Etages | self.Cabine.Position.Numero < Numero and Ap
```

```
pel_descente ] and not EXISTS[ Etages | self.Cabine.Position.
Numero < Numero and (Appel_montee or isin(self.Cabine.Etages_
Arret,current))] and not Cabine.Position.Appel_descente and n
ot isin(Cabine.Etages_Arret,Cabine.Position)
Changer_Phase.NouvellePhase:=  TPhase$PREPAREDOWN
```

Transition (12) with Changer_Phase

?:

```
not EXISTS[Etages | Appel_montee or Appel_descente or isin(se
lf.Cabine.Etages_Arret, current)] or (EXISTS[Etages | self.Ca
bine.Position.Numero < Numero and Appel_descente] and not EXI
STS[Etages| Appel_montee or isin(self.Cabine.Etages_Arret,cur
rent)] and not EXISTS[Etages| self.Cabine.Position.Numero > N
umero and Appel_descente] and not Cabine.Position.Appel_desce
nte)
Changer_Phase.NouvellePhase:=  TPhase$STABLE
```

Transition (11) with Changer_Phase

?:

```
not EXISTS[Etages | Appel_montee or Appel_descente or isin(se
lf.Cabine.Etages_Arret, current)] or (EXISTS[Etages | self.Ca
bine.Position.Numero > Numero and Appel_montee] and not EXIST
S[Etages| Appel_descente or isin(self.Cabine.Etages_Arret,cur
rent)] and not EXISTS[Etages| self.Cabine.Position.Numero < N
umero and Appel_montee] and not Cabine.Position.Appel_montee)
Changer_Phase.NouvellePhase:=  TPhase$STABLE
```

Transition (6) with Changer_Phase

?:

```
EXISTS[Etages | self.Cabine.Position.Numero > Numero and (App
el_descente or isin(self.Cabine.Etages_Arret,current))]
Changer_Phase.NouvellePhase:=  TPhase$DOWN
```

Transition (7) with Changer_Phase

?:

```
EXISTS[Etages | self.Cabine.Position.Numero > Numero and Appe
l_montee ]
Changer_Phase.NouvellePhase:=  TPhase$PREPAREUP
```

Transition (8) with Changer_Phase

?:

```
EXISTS[Etages | self.Cabine.Position.Numero < Numero and Appe
l_descente ]
Changer_Phase.NouvellePhase:=  TPhase$PREPAREDOWN
```

Transition (9) with Changer_Phase

?:

```
        not EXISTS[Etages | self.Cabine.Position.Numero > Numero and
        Appel_montee]
      Changer_Phase.NouvellePhase:=  TPhase$UP
```

Transition (10) with Changer_Phase

?:

```
        not EXISTS[Etages | self.Cabine.Position.Numero < Numero and
        Appel_descente]
      Changer_Phase.NouvellePhase:=  TPhase$DOWN
```

Transition (5) with Changer_Phase

?:

```
        EXISTS[Etages | self.Cabine.Position.Numero < Numero and (App
        el_montee or isin(self.Cabine.Etages_Arret,current))]
      Changer_Phase.NouvellePhase:=  TPhase$UP
```

Transition (26) with Commander_arret

?:

```
        Cabine.Position.Appel_montee or isin(Cabine.Etages_Arret,Cabi
        ne.Position)
```

Transition (17) with Commander_descente

?:

```
        EXISTS[Etages | self.Cabine.Position.Numero > Numero and Appe
        l_montee]
```

Transition (20) with Commander_descente

?:

```
        not Cabine.Position.Appel_descente and not isin(Cabine.Etages
        _Arret,Cabine.Position)
```

Transition (18) with Commander_montee

?:

```
        EXISTS[Etages | self.Cabine.Position.Numero < Numero and Appe
        l_descente]
```

Transition (19) with Commander_montee

?:

```
        not Cabine.Position.Appel_montee and not isin(Cabine.Etages_A
        rret,Cabine.Position)
```

Transition (33) with Fermer_porte

?: `Phase = TPhase$UP`

```
        Fermer_porte.Etage:=  Cabine.Position
```

Transition (36) with Fermer_porte

```
        ?: Phase = TPhase$PREPAREUP
        Fermer_porte.Etage:=  Cabine.Position
```

Transition (35) with Fermer_porte

```
        ?: Phase = TPhase$PREPAREDOWN
        Fermer_porte.Etage:=  Cabine.Position
```

Transition (34) with Fermer_porte

```
        ?: Phase = TPhase$DOWN
        Fermer_porte.Etage:=  Cabine.Position
```

Transition (21) with Ouvrir_porte

```
        ?:
           Cabine.Position.Appel_montee or Cabine.Position.Appel_descent
           e
        Ouvrir_porte.Position:=  Cabine.Position
```

Transition (30) with Ouvrir_porte

```
        Ouvrir_porte.Position:=  Cabine.Position
```

Transition (65) with Prevenir

```
        ?:
           (Panne_a_venir or Cabine.Panne_a_venir or moteur.Panne_a_veni
           r or EXISTS[Etages | Panne_a_venir]) and not Prevenu
```

Transition (49) with Prevenir

```
        ?:
           (Panne_a_venir or Cabine.Panne_a_venir or moteur.Panne_a_veni
           r or EXISTS[Etages | Panne_a_venir]) and not Prevenu
```

Transition (57) with Prevenir

```
        ?:
           (Panne_a_venir or Cabine.Panne_a_venir or moteur.Panne_a_veni
           r or EXISTS[Etages | Panne_a_venir]) and not Prevenu
```

Transition (37) with Prevenir

```
        ?:
           (Panne_a_venir or Cabine.Panne_a_venir or moteur.Panne_a_veni
           r or EXISTS[Etages | Panne_a_venir]) and not Prevenu
```

Transition (52) with Prevenir

?:

```
(Panne_a_venir or Cabine.Panne_a_venir or moteur.Panne_a_veni
r or EXISTS[Etages | Panne_a_venir]) and not Prevenu
```

Transition (45) with Prevenir

?:

```
(Panne_a_venir or Cabine.Panne_a_venir or moteur.Panne_a_veni
r or EXISTS[Etages | Panne_a_venir]) and not Prevenu
```

Transition (41) with Prevenir

?:

```
(Panne_a_venir or Cabine.Panne_a_venir or moteur.Panne_a_veni
r or EXISTS[Etages | Panne_a_venir]) and not Prevenu
```

Transition (61) with Prevenir

?:

```
(Panne_a_venir or Cabine.Panne_a_venir or moteur.Panne_a_veni
r or EXISTS[Etages | Panne_a_venir]) and not Prevenu
```

Transition (3) with Referencer

?: `EXISTS [ Moteur | TRUE ]`

Transition (50) with Verifier

?:

```
not Panne_a_venir and not Cabine.Panne_a_venir and not moteur
.Panne_a_venir and not EXISTS[Etages | Panne_a_venir] and Pre
venu
```

Transition (38) with Verifier

?:

```
not Panne_a_venir and not Cabine.Panne_a_venir and not moteur
.Panne_a_venir and not EXISTS[Etages | Panne_a_venir] and Pre
venu
```

Transition (46) with Verifier

?:

```
not Panne_a_venir and not Cabine.Panne_a_venir and not moteur
.Panne_a_venir and not EXISTS[Etages | Panne_a_venir] and Pre
venu
```

Transition (51) with Verifier
?:

```
not Panne_a_venir and not Cabine.Panne_a_venir and not moteur
.Panne_a_venir and not EXISTS[Etages | Panne_a_venir] and Pre
venu
```

Transition (62) with Verifier
?:

```
not Panne_a_venir and not Cabine.Panne_a_venir and not moteur
.Panne_a_venir and not EXISTS[Etages | Panne_a_venir] and Pre
venu
```

Transition (42) with Verifier
?:

```
not Panne_a_venir and not Cabine.Panne_a_venir and not moteur
.Panne_a_venir and not EXISTS[Etages | Panne_a_venir] and Pre
venu
```

Transition (66) with Verifier
?:

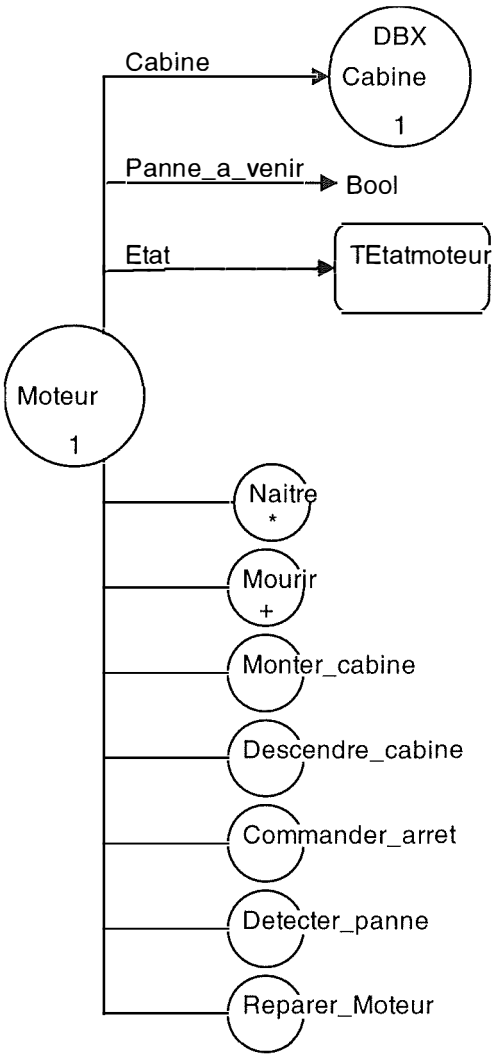
```
not Panne_a_venir and not Cabine.Panne_a_venir and not moteur
.Panne_a_venir and not EXISTS[Etages | Panne_a_venir] and Pre
venu
```

Transition (58) with Verifier
?:

```
not Panne_a_venir and not Cabine.Panne_a_venir and not moteur
.Panne_a_venir and not EXISTS[Etages | Panne_a_venir] and Pre
venu
```

Transition (4) with VerifierExistence
?: `not EXISTS [ Moteur | TRUE ]`

Contents: Declaration Diagram **Class:** Ascenceur:Moteur	**Date:** Wed Aug 28 1996 13:17:13 **By:** Unknown



Contents: Local Class Declaration	**Date:** Wed Aug 28 1996 13:17:13
Class: Moteur	**By:** Unknown

Single Class Moteur

Interactions

to Cabine **using**
- Monter_cabine
- Descendre_cabine

Attributes

Cabine	:	Cabine
Etat	:	TEtatmoteur
Panne_a_venir	:	Bool

Operations

*Naitre
Commander_arret
Descendre_cabine
Detecter_panne
Monter_cabine
Reparer_Moteur
+Mourir

End Class Moteur

Operation Naitre

Updates

```
Cabine  :=  ONE[Cabine | TRUE]
Etat    :=  TEtatmoteur$ARRETE
Panne_a_venir  :=  FALSE
```

Operation Commander_arret

Updates

```
Etat    :=  TEtatmoteur$ARRETE
```

Operation Descendre_cabine

Updates

```
Etat    :=  TEtatmoteur$DESCENDANT
```

Calls

Cabine.Descendre_cabine **of object identified by** Cabine

Operation Detecter_panne

Updates

```
Panne_a_venir  :=  TRUE
```

Operation Monter_cabine

Updates

```
Etat    :=  TEtatmoteur$MONTANT
```

Calls

Cabine.Monter_cabine **of object identified by** Cabine

Operation Reparer_Moteur

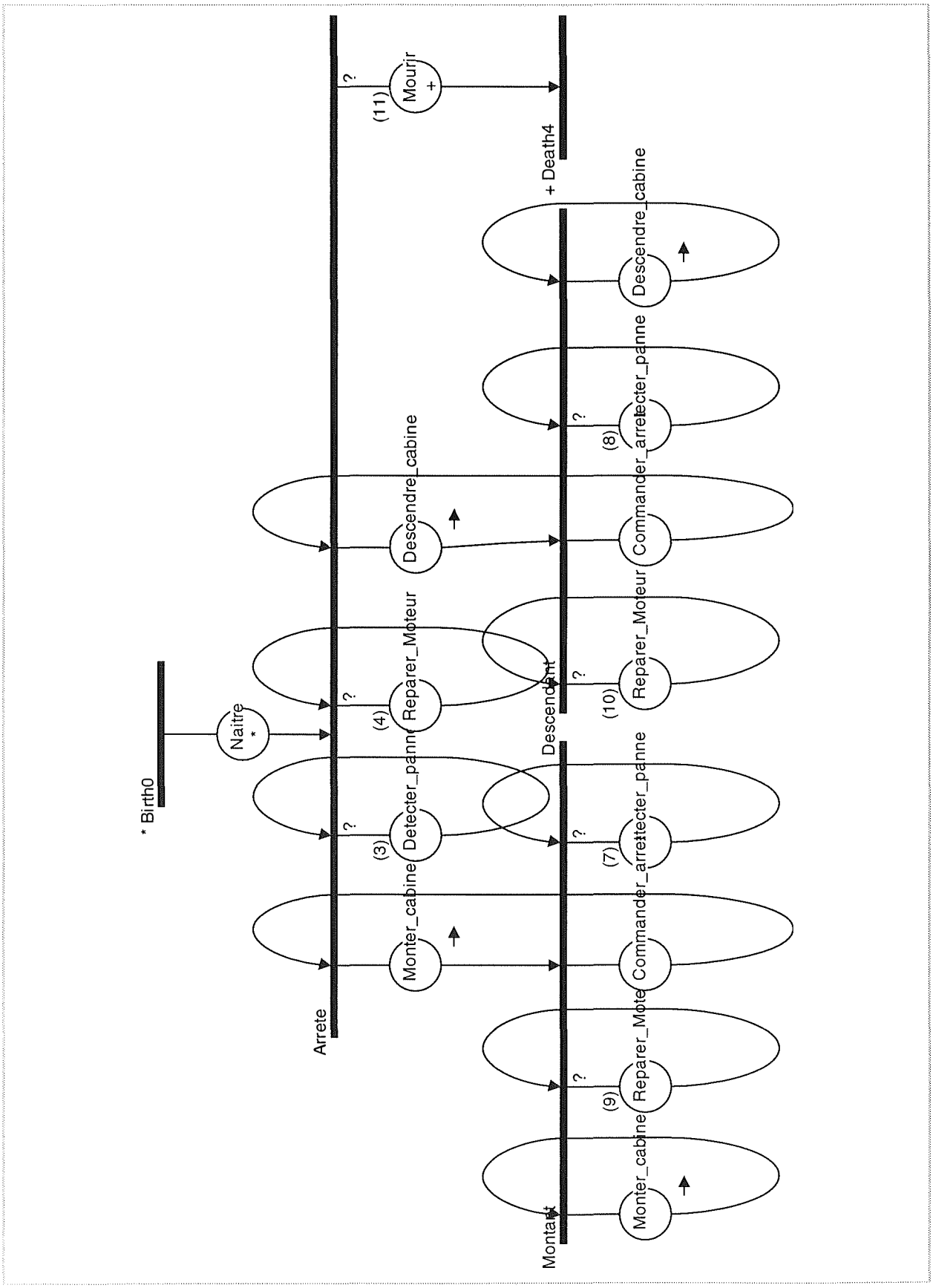
Updates

```
Panne_a_venir := FALSE
```

Operation Mourir

No Properties

Contents: Behavior Diagram	Date: Wed Aug 28 1996 13:17:13
Class: Ascenceur:Moteur	By: Unknown



Contents: Behavior Description	Date: Wed Aug 28 1996 13:17:13
Class: Ascenceur:Moteur	By: Unknown

Behavior of class Moteur

Birth State Birth0 **:**

⟶ *Naitre ⟶ Arrete

State Arrete **:**

⟶ +Mourir ⟶ Death4
?: not Panne_a_venir

⟶ Monter_cabine ⟶ Montant

⟶ Descendre_cabine ⟶ Descendant

⟶ Detecter_panne ⟶ Arrete
?: not Panne_a_venir

⟶ Reparer_Moteur ⟶ Arrete
?: Panne_a_venir

State Descendant **:**

⟶ Descendre_cabine ⟶ Descendant

⟶ Detecter_panne ⟶ Descendant
?: not Panne_a_venir

⟶ Reparer_Moteur ⟶ Descendant
?: Panne_a_venir

⟶ Commander_arret ⟶ Arrete

State Montant **:**

⟶ Monter_cabine ⟶ Montant

⟶ Detecter_panne ⟶ Montant
?: not Panne_a_venir

⟶ Reparer_Moteur ⟶ Montant
?: Panne_a_venir

⟶ Commander_arret ⟶ Arrete

Death State Death4 **:**

Contents: Behavior Description **Class:** Ascenceur:Moteur	**Date:** Wed Aug 28 1996 13:17:13 **By:** Unknown

Expressions in behavior of class `Moteur`

Transition (7) with `Detecter_panne`
?: `not Panne_a_venir`

Transition (3) with `Detecter_panne`
?: `not Panne_a_venir`

Transition (8) with `Detecter_panne`
?: `not Panne_a_venir`

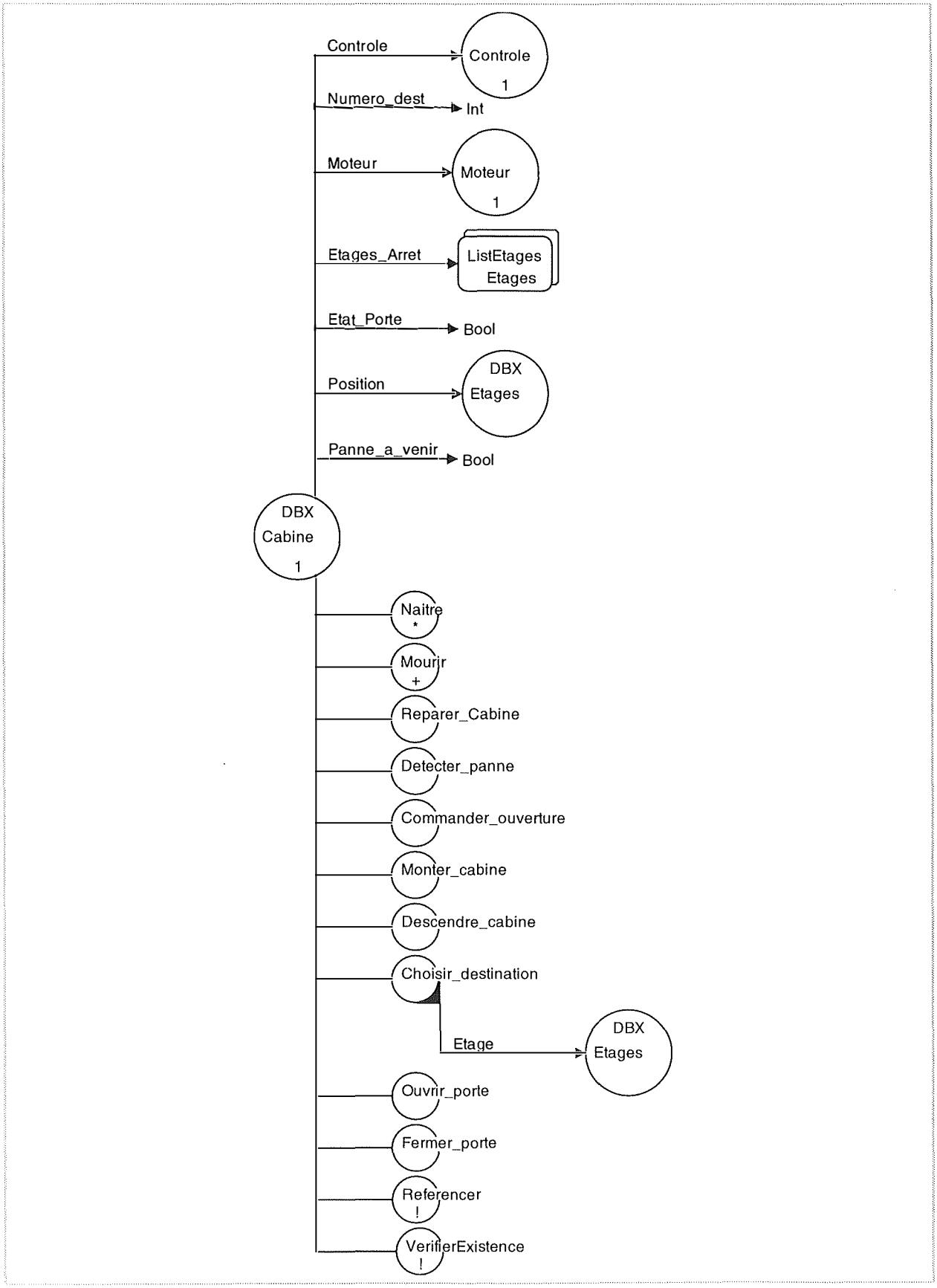
Transition (10) with `Reparer_Moteur`
?: `Panne_a_venir`

Transition (4) with `Reparer_Moteur`
?: `Panne_a_venir`

Transition (9) with `Reparer_Moteur`
?: `Panne_a_venir`

Transition (11) with `Mourir`
?: `not Panne_a_venir`

Contents: Declaration Diagram **Class:** Ascenceur:Cabine	**Date:** Wed Aug 28 1996 13:17:13 **By:** Unknown



Contents: Local Class Declaration	Date: Wed Aug 28 1996 13:17:13
Class: Cabine	By: Unknown

Single DBX Class Cabine

Interactions

to Controle **using**
- Commander_ouverture

Attributes

Controle	:	Controle
Etages_Arret	:	ListEtages
Etat_Porte	:	Bool
Moteur	:	Moteur
Numero_dest	:	Int
Panne_a_venir	:	Bool
Position	:	Etages

Operations

*Naitre

Choisir_destination
Etage : Etages

Commander_ouverture
Descendre_cabine
Detecter_panne
Fermer_porte
Monter_cabine
Ouvrir_porte
Reparer_Cabine
!Referencer
!VerifierExistence
+Mourir

End Class Cabine

Operation Naitre

Updates

```
Etat_Porte     :=  FALSE
Panne_a_venir  :=  FALSE
```

Operation Choisir_destination

Parameters

```
Etage                                  :  Etages
```

Updates

```
Etages_Arret   :=
    append(Etages_Arret,Choisir_destination.Etage)
```

Operation Commander_ouverture

Calls

Controle.Attendre_5s_passif **of object identified by** Controle
?: Controle.Phase <> TPhase$STABLE

Controle.Commander_ouverture **of object identified by** Controle
?: Controle.Phase = TPhase$STABLE

Operation Descendre_cabine

Updates

```
Position    :=
    ONE[Etages | Numero = self.Position.Numero - 1]
```

Operation Detecter_panne

Updates

```
Panne_a_venir  :=  TRUE
```

Operation Fermer_porte

Updates

```
Etat_Porte     :=  FALSE
```

Operation Monter_cabine

Updates

```
Position   :=
   ONE[Etages | Numero = self.Position.Numero + 1]
```

Operation Ouvrir_porte

Updates

```
Etat_Porte     :=  TRUE
Etages_Arret   :=  remove(Etages_Arret,Position)
```

Operation Reparer_Cabine

Updates

```
Panne_a_venir  :=  FALSE
```

Operation Referencer

Updates

```
Position   :=  ONE [ Etages | Numero = 0 ]
Controle   :=  ONE[ Controle | TRUE ]
Moteur  :=  ONE[ Moteur | TRUE ]
```

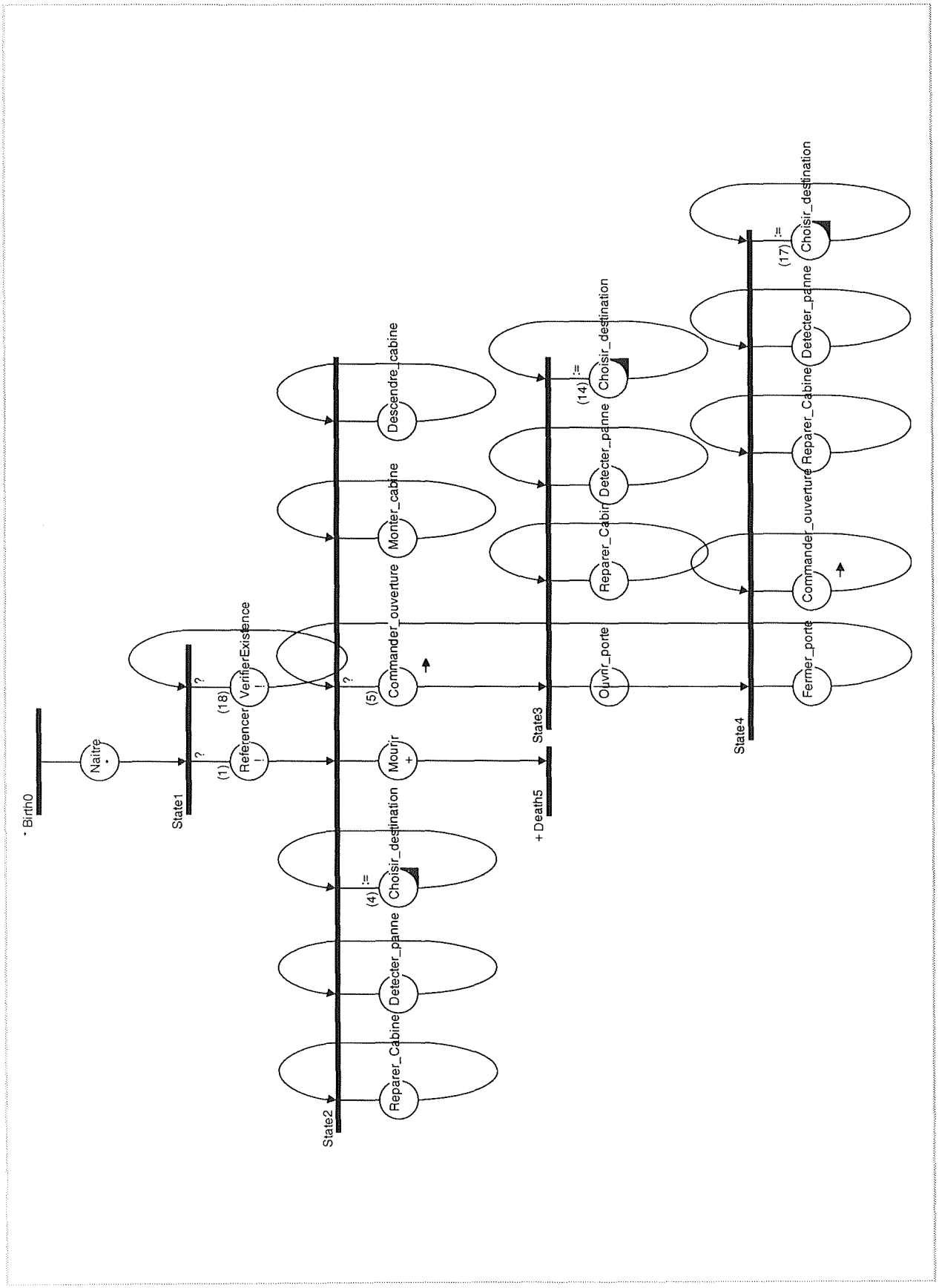
Operation VerifierExistence

No Properties

Operation Mourir

No Properties

Contents: Behavior Diagram	**Date:** Wed Aug 28 1996 13:17:13
Class: Ascenceur:Cabine	**By:** Unknown



Behavior of class Cabine

Birth State Birth0 **:**

⟶▷ *Naitre ⟶▷ State1

State State1 **:**

⟶▷ !VerifierExistence ⟶▷ State1
?: not EXISTS[Moteur | true]

⟶▷ !Referencer ⟶▷ State2
?: EXISTS[Moteur | true]

State State2 **:**

⟶▷ Monter_cabine ⟶▷ State2

⟶▷ Descendre_cabine ⟶▷ State2

⟶▷ +Mourir ⟶▷ Death5

⟶▷ Reparer_Cabine ⟶▷ State2

⟶▷ Detecter_panne ⟶▷ State2

⟶▷ Choisir_destination ⟶▷ State2
Choisir_destination.Etage:=
ONE[Etages | Numero = self.Numero_dest]

⟶▷ Commander_ouverture ⟶▷ State3
?: Moteur.Etat = TEtatmoteur$ARRETE

State State3 **:**

⟶▷ Reparer_Cabine ⟶▷ State3

⟶▷ Detecter_panne ⟶▷ State3

⟶▷ Choisir_destination ⟶▷ State3
Choisir_destination.Etage:=
ONE[Etages | Numero = self.Numero_dest]

⟶▷ Ouvrir_porte ⟶▷ State4

State State4 **:**

⟶▷ Reparer_Cabine ⟶▷ State4

⟶▷ Detecter_panne ⟶▷ State4

⟶▷ Choisir_destination ⟶▷ State4

```
Choisir_destination.Etage:=
    ONE[Etages | Numero = self.Numero_dest ]
```

⟶▷ Fermer_porte ⟶▷ State2

⟶▷ Commander_ouverture ⟶▷ State4

Death State Death5 **:**

Expressions in behavior of class `Cabine`

Transition (4) with `Choisir_destination`

```
Choisir_destination.Etage:=
    ONE[Etages | Numero = self.Numero_dest ]
```

Transition (17) with `Choisir_destination`

```
Choisir_destination.Etage:=
    ONE[Etages | Numero = self.Numero_dest ]
```

Transition (14) with `Choisir_destination`

```
Choisir_destination.Etage:=
    ONE[Etages | Numero = self.Numero_dest ]
```

Transition (5) with `Commander_ouverture`

?: `Moteur.Etat = TEtatmoteur$ARRETE`

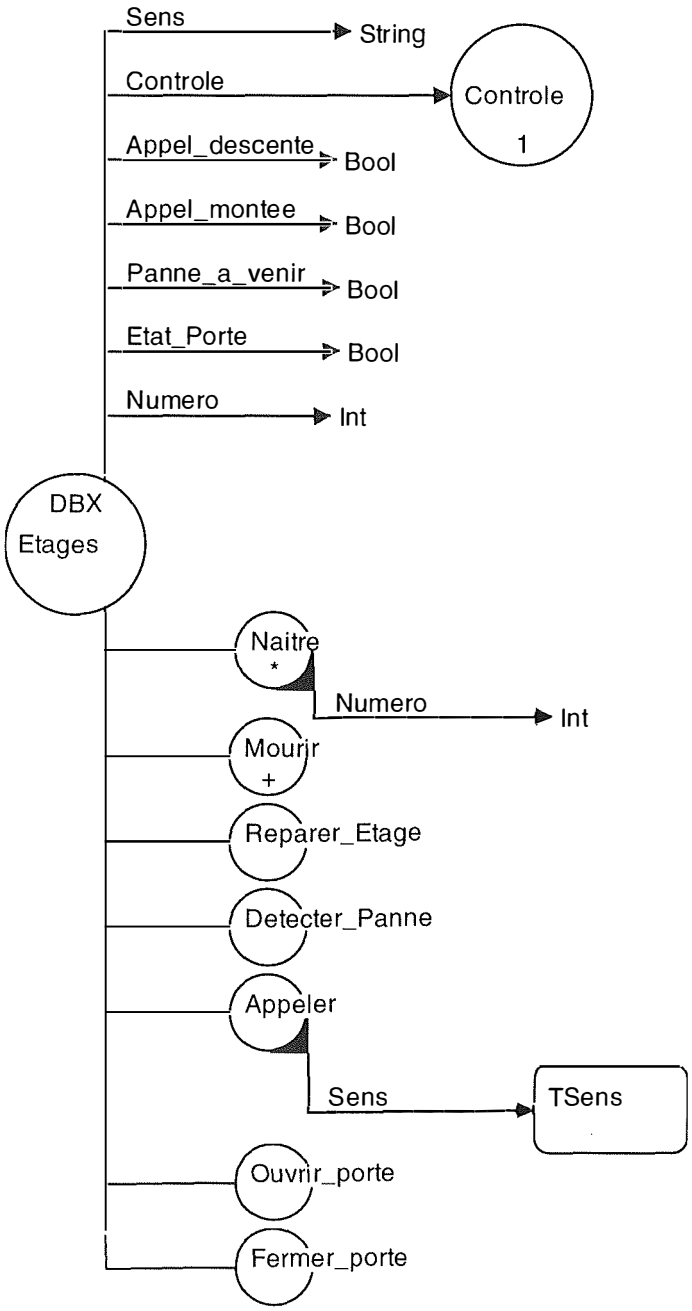
Transition (1) with `Referencer`

?: `EXISTS[Moteur | true]`

Transition (18) with `VerifierExistence`

?: `not EXISTS[Moteur | true]`

Contents: Declaration Diagram **Class:** Ascenceur:Etages	**Date:** Wed Aug 28 1996 13:17:13 **By:** Unknown



Contents: Local Class Declaration	Date: Wed Aug 28 1996 13:17:13
Class: Etages	By: Unknown

DBX Class Etages

Attributes

```
Appel_descente                  : Bool
Appel_montee                    : Bool
Controle                        : Controle
Etat_Porte                      : Bool
Numero                          : Int
Panne_a_venir                   : Bool
Sens                            : String
```

Operations

```
*Naitre
   Numero                       : Int

Appeler
   Sens                         : TSens

Detecter_Panne
Fermer_porte
Ouvrir_porte
Reparer_Etage
+Mourir
```

End Class Etages

Operation Naitre

Parameters

```
Numero                                  :  Int
```

Updates

```
Controle    :=  ONE[Controle | TRUE ]
Numero  :=  Naitre.Numero
```

Operation Appeler

Parameters

```
Sens                                    :  TSens
```

Updates

```
Appel_descente    :=
    if(Appeler.Sens=TSens$DESCENTE and Numero >-1,TRUE, FAL
   SE)
Appel_montee   :=
    if(Appeler.Sens=TSens$MONTEE and Numero < 4,TRUE, FALSE
   )
```

Operation Detecter_Panne

Updates

```
Panne_a_venir  :=  TRUE
```

Operation Fermer_porte

Updates

```
Etat_Porte     :=  FALSE
```

Operation Ouvrir_porte

Updates

```
Etat_Porte     :=  TRUE
Appel_descente    :=
    if(Controle.Phase = TPhase$DOWN,FALSE,Appel_descente)
Appel_montee   :=
    if(Controle.Phase = TPhase$UP, FALSE, Appel_montee)
```

Operation Reparer_Etage

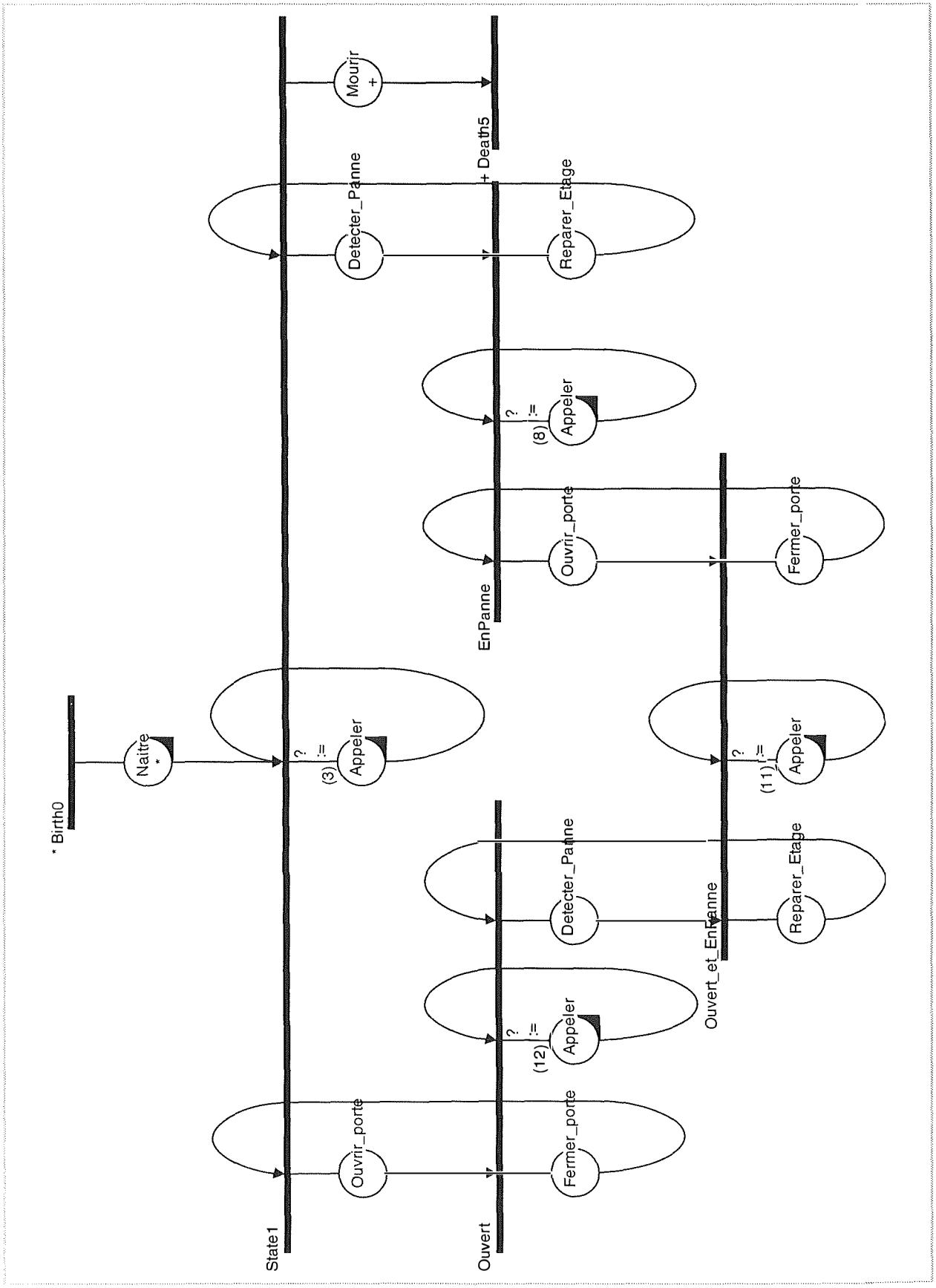
Updates

```
Panne_a_venir  :=  FALSE
```

Operation Mourir

No Properties

Contents: Behavior Diagram **Class:** Ascenceur:Etages	**Date:** Wed Aug 281 996 131 7:13 **By:** Unknown



Behavior of class Etages

Birth State Birth0 **:**

⟶ *Naitre ⟶ State1

State EnPanne **:**

⟶ Ouvrir_porte ⟶ Ouvert et EnPanne

⟶ Appeler ⟶ EnPanne

?:

(Sens = "MONTEE" and not Appel_montee) or (Sens = "DESCENTE" and not Appel_descente)

```
Appeler.Sens:=
    if(Sens = "MONTEE", TSens$MONTEE, TSens$DESCENTE)
```

⟶ Reparer_Etage ⟶ State1

State Ouvert **:**

⟶ Appeler ⟶ Ouvert

?:

(Sens = "MONTEE" and not Appel_montee) or (Sens = "DESCENTE" and not Appel_descente)

```
Appeler.Sens:=
    if(Sens = "MONTEE", TSens$MONTEE, TSens$DESCENTE)
```

⟶ Fermer_porte ⟶ State1

⟶ Detecter_Panne ⟶ Ouvert et EnPanne

State Ouvert et EnPanne **:**

⟶ Fermer_porte ⟶ EnPanne

⟶ Reparer_Etage ⟶ Ouvert

⟶ Appeler ⟶ Ouvert et EnPanne

?:

(Sens = "MONTEE" and not Appel_montee) or (Sens = "DESCENTE" and not Appel_descente)

```
Appeler.Sens:=
    if(Sens = "MONTEE", TSens$MONTEE, TSens$DESCENTE)
```

State State1 **:**

⟶ +Mourir ⟶ Death5

⟶▷ Ouvrir_porte ⟶▷ Ouvert

⟶▷ Appeler ⟶▷ State1

?:

```
(Sens = "MONTEE" and not Appel_montee) or (Sens = "DESCENTE"
and not Appel_descente)
```

```
Appeler.Sens:=
    if(Sens = "MONTEE", TSens$MONTEE, TSens$DESCENTE)
```

⟶▷ Detecter_Panne ⟶▷ EnPanne

Death State Death5 **:**

Expressions in behavior of class `Etages`

Transition (12) with `Appeler`

?:

```
(Sens = "MONTEE" and not Appel_montee) or (Sens = "DESCENTE"
and not Appel_descente)
```

```
Appeler.Sens:=
    if(Sens = "MONTEE", TSens$MONTEE, TSens$DESCENTE)
```

Transition (11) with `Appeler`

?:

```
(Sens = "MONTEE" and not Appel_montee) or (Sens = "DESCENTE"
and not Appel_descente)
```

```
Appeler.Sens:=
    if(Sens = "MONTEE", TSens$MONTEE, TSens$DESCENTE)
```

Transition (8) with `Appeler`

?:

```
(Sens = "MONTEE" and not Appel_montee) or (Sens = "DESCENTE"
and not Appel_descente)
```

```
Appeler.Sens:=
    if(Sens = "MONTEE", TSens$MONTEE, TSens$DESCENTE)
```

Transition (3) with `Appeler`

?:

```
(Sens = "MONTEE" and not Appel_montee) or (Sens = "DESCENTE"
and not Appel_descente)
```

```
Appeler.Sens:=
    if(Sens = "MONTEE", TSens$MONTEE, TSens$DESCENTE)
```